MRAMFS: A Compressing File System for Byte-Addressable Non-Volatile RAM

Technical Report UCSC-SSRC-11-02
March 2011

Nathan Keir Edel
nate@cs.ucsc.edu

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

Filed as a MS thesis in the Computer Science Department of the University of California, Santa Cruz in March 2011. Some of the material in this thesis was published in MASCOTS 2004 and SPECTS 2003.

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

MRAMFS: A COMPRESSING FILE SYSTEM FOR BYTE-ADDRESSABLE NON-VOLATILE RAM

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nathan Keir Edel

March 2011

The Thesis of Nathan Keir Edel
is approved:

Professor Ethan L. Miller, Chair

Professor Darrell D. E. Long

Dr. Kevin Greenan

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

MRAMFS: a compressing file system for byte-addressable non-volatile RAM

by

Nathan Keir Edel

File systems combining block storage with non-volatile RAM (NVRAM) allow large improvements in file system performance. However, current technology limits low-cost use of NVRAM to relatively low capacities; we examined in-memory compression methods which allow for significantly more efficient utilization of this limited resource.

The first phase of the study measured the compressibility of these objects for a set of representative file systems. We found that inodes are compressible by at least 76–90% at a rate of 270–900 thousand inodes per second for the best algorithms. Files in the range of 4–128 KB were compressible of 40–60% at rates of 20–40 megabytes per second.

In the second phase of the study, we developed a prototype in-memory file system which utilizes data compression on inodes, and attempted preliminary support for compression of file blocks. This file system, *mramfs*, allowed for the examination of data structures tuned for storage efficiency in non-volatile memory. It showed that for metadata operations, inode compression does not significantly impact performance, while significantly reducing the space used. It also showed that a naive implementation of block-based file compression does not perform acceptably either in terms of speed or compression achieved.

To Marie,

without whose patience and encouragement

this would never have been possible.

Acknowledgments

I would like to thank the other members of the Storage Systems Research Center for their help and feedback in preparing the original conference papers upon which this thesis is based. [22, 23]

In addition, I would like to individually thank:

Ethan Miller as my graduate advisor and committee chair, and specifically for suggesting the ideas of inode compression and using gamma compression for it; and for shepherding the original pair of conference papers.

Darrell Long and Kevin Greenan as members of my committee.

Karl Brandt for coding the file compression test harness and his collaboration on the design of file compression tests. Any errors of analysis are purely my own.

Kristal Pollack, Phil White, and Andrew Stitt for their help with inode data collection.

Deepa Tuteja for her suggestions on benchmarking the prototype file system and feedback on the MASCOTS paper.

Tim Bisson as a sounding board throughout the research phases of this project.

The original 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.

Chapter 1

Introduction

File systems implemented in RAM or NVRAM offer much greater performance than disk based file systems, especially for random accesses and metadata operations. However, capacity constraints and the volatility of conventional RAM limit their general utility. Hybrid 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 (BBDRAM) and flash memory—as well as emerging technologies such as magnetic RAM (MRAM) and phase-change RAM (PCRAM).

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 [58]. Accesses to small objects are primarily limited by time to first byte, making RAM-like technologies more 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.

Despite claims to the contrary [74], with the exception of NAND flash, non-volatile memory capacities can be expected to be limited for the foreseeable future, and

NAND-flash remains a form of block storage and still relatively expensive compared to disk. While byte-addressable NVRAM prices may approach those of flash in the long run, at present they are niche or emerging technologies which can be expected to be limited in capacity in the near term.[1]

Assuming non-volatile memory capacities 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 is of particular interest because of the rapid improvement in processor speeds relative to the slow improvement of storage bandwidth and latency [76].

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. While there has been some interest in compression of caches in purely volatile memory both in general [20, 40, 55, 17, 31] and specifically for disk caches[12], limits on NVRAM-based or hybrid file systems make compression a particularly attractive tool for reducing capacity requirements and system cost.

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) [77, 24].

1.1 This Study

We explored the potential space savings and performance cost of compression for metadata and small files. In the first phase, we examined compression mechanisms

[1] In earlier work, we made the comparison to the cost of DRAM, which was then still cheaper than flash. Flash is now considerably cheaper than DRAM, but still pricier than disk. There is relatively little technical reason BBDRAM should be much pricier than conventional volatile DRAM—differing in the component cost of the battery and controller—but in practice in the consumer and workstation market it has remained limited to niche products, and its cost in enterprise-level products remains very high.

without regard to specific design decisions for a disk/NVRAM hybrid file system. We focused on static compression methods for metadata and stream compression for file data. This was based on the assumption that static compression would be better suited to independently compressing the very small blocks of data in metadata, while stream compressors would better handle the unknown data types for file data.

In the second phase, we developed a prototype file system for Linux, *mramfs*, in order to explore the costs and benefits of metadata and small-file compression. It differs from prior file systems in that it is not tied to volatile main memory through tight integration with the VFS caches like `ramfs` or `tmpfs` [64], 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 an unjournaled disk file system such as `ext2` on a RAM disk or emulated block device. However, with compression and some structures tuned specifically for random access memory, it offers greater space efficiency and potentially better performance. It could also serve as the basis for a future disk/NVRAM hybrid system along the lines proposed for HeRMES [50].

Both phases of the study made certain assumptions about the target system for which a final file system might be developed. In particular, it assumes the range of systems and applications to be supported were intended for PC-class systems running Linux or a similar Unix-like operating system; we did not consider whether it would be appropriate for mobile or embedded systems.

While we did not assume the use of any particular kind of non-volatile memory technology, it does assume that the NVRAM is randomly accessible at byte (or word) boundaries for both reads and writes. Our performance analysis also assumes that the NVRAM has predictable (but not necessarily symmetric) random access performance for both reads and writes. Additionally, the prototype assumes that the NVRAM can be mapped directly into the system address space, although relaxing this requirement would add only slightly to complexity.

It should also be 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]

Although the compression techniques we propose could be used as part of a file system for any sort of flash memory, our system would need to be restructured significantly to be useful. All forms of flash memory have very distinctive characteristics that present separate challenges to file system design [21, 68, 79], and these issues would present similar challenges to designs for hybrid file systems incorporating flash memory. Other studies have suggested that using a log structure is preferable for a flash-specific file system, among other reasons because it lends itself to efficient wear-leveling [59, 79, 78]. One proposed use for non-flash NVRAM in such a hybrid flash/nvram system is to make the log structures themselves more efficient [36]. File data compression in a log structured file system has been a subject of research [8], and is implemented in some file systems for embedded flash.

[2]With present processors, that may apply to disk in some cases as well—note the very positive reports on compression performance on `zfs` [44] and `btrfs` [43].

Chapter 2

Related Work

The use of non-volatile memory for file systems is not new; Wu and Zwaenepoel [79] and Kawaguchi, *et al.* [41] presented designs for flash memory-based file systems. Existing flash storage may either use a flash translation layer (FTL)—which can be implemented in either software or hardware, and which exposes the flash device as if it were a disk to be used by a conventional file system—or any one of a number of dedicated file systems for flash.

Dedicated file systems optimized to run on flash memory (whether generally or specifically for NAND or NOR variants) include the Microsoft Flash File System [21, 45], `JFFS2` [78] and YAFFS. More recently, the authors of `BPFS` [11] suggested a design for tree-based filesytem adapted to byte-addressable NVRAM (and PCRAM in particular.)

Of particular note, `JFFS2` is a log-structured file system [59] optimized for flash memory usage that does support compression of data and metadata, but it cannot support hybrid 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 [8, 80], and compression has been proposed as an extension to the commonly used `ext2` file system on Linux [73]. Compression has also been used at the flash translation layer level [33].

None of the older flash specific file systems have taken a hybrid approach with disk, and most have been aimed at embedded or mobile systems rather than general purpose computing. Some more recent embedded file systems have used a

hybrid flash plus NVRAM approach, with FeRAM (as in the case of FRASH [36]) or PCRAM [42, 57, 26] as the non-flash component. Other more recent systems explored the combined use of flash SSD with conventional disk. HybridFS [66] used SSD to store metadata with disk for file data, Combo Drive [54] focussed on data access patterns to divide data between SSD and disk, while Griffin [65] suggested the use of disk as a logging write cache for SSD.

Over the past decade, there had been a renewed interest in hybrid disk/NVRAM file system, particularly as flash memory has dropped in price and alternative technologies such as MRAM [5, 71, 83], FeRAM [49], and phase change memory (PCRAM) [18, 57] appeared to be coming closer to being in production. The HeRMES file system [50] and the Conquest file system [74] are examples of hybrid disk/NVRAM file systems in academe. However, the two systems make 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 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 considerable number of studies of the distribution of file sizes, and file lifetimes [2, 60, 58, 25, 69]. There has also been some discussion of the distribution of file ownership and permissions as it relates to file system security [37, 56].

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 [20]. This idea has been expanded upon in several directions; Wilson, Kaplan,

and Smaragdakis evaluated the use of different compression mechanisms for memory data [40, 76], and Cortes, *et al.* evaluated the performance of using such techniques on a modern system [16]. As of 2003, there was an ongoing effort to implement a compressed page cache on Linux [17]; this seems to have been superceded by an implementation of a compressed block driver for swap which is (since 2.6.33 in late 2009) included as a staging driver in the mainline Linux kernel [31].

A number of compression mechanisms could be used to compress metadata, including any of the block or stream mechanisms evaluated by Wilson, *et al.* [76] and used in the Linux-Compressed project [17]. However, simpler mechanisms such as Huffman coding using a precomputed tree [13], gamma compression [77], and other prefix encodings [77] can all be used to good effect without the same degree of runtime processing overhead.

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 [32]. 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. Recently, NVRAM buffering in conjunction with flash has been shown to be advantageous for hot writes [26] and for speeding up operations on log-structured flash[19, 36]. It has also been suggested that an NVRAM write cache could help with power consumption by allowing disks to remain in a low-power/spun-down state [3]. It is conceivable that compressing written data into a buffer would effectively increase its size.

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` [46]), and many commercial UNIX variants [64]. There is an early effort to extend the swap specific compressed cache on Linux to a general compressed ramdisk [12, 31].

Battery-backed RAM has in the past been frequently used in mobile devices,

such as those running older versions of Windows CE or the Palm OS. This has largely been replaced by flash based storage. Douglis, *et al.* [21] studied storage alternatives for mobile computers, including two types of flash memory. They noted that flash memory as it existed at the time was slow, particularly for writes. This has changed; fast commodity and enterprise solid state disks exist as drop-in replacements for SATA hard drives. These offer faster performance than disks in most dimensions, although challenges remain in other areas—most notably, costs remain quite high relative to disk.[1] There have also been interest in disk-flash hybrid systems, both in academia and industry; there have been attempts to use hybrid disks with flash on board [4, 63] and to use a separate SSD or flash card as a a cache to speed up reads [15, 53].

In addition, NAND flash—the primary form of flash used for bulk storage — has a block interface and is not byte-addressable for either reads or writes. Many of the same arguments for a hybrid disk-NVRAM file system apply to a hybrid flash-NVRAM file system, and such systems have been developed. [36]

While this prototype could be used with any fast byte-addressable form of NVRAM, there has been significant interest in MRAM specifically [5, 71, 83]. There were a number of recent technical advances around 2004, including a fast (35ns) 4 Mbit part discussed by Motorola [51] and eventually released (now sold by Everspin, after divestiture by Motorola and Freescale in turn.) While this has had market availability for several years, and largest broadly available capacity of 4 megabits either as 512K bytes or 256Kx16-bit has remained constant, at a fairly consistent price—around $25–30 [14] as of early 2010; a 16Mbit (1Mx16-bit) part was announced by Everspin in 2010 [70]. This is several orders of magnitude smaller and pricier than DRAM; it remains unknown how quickly this will evolve past a niche product with capacities too small for significant mass storage use. FeRAM (Ferroelectric RAM) is also available in slightly higher densities; the FRASH embedded file system uses a 64Mbit FeRAM unit although it was not clear whether this was commercially available or a research sample [36]; another non-storage study used smaller FeRAM parts (in an embedded web server) at around the same cost and density as commercial MRAM. [34]

Practical battery-backed DRAM and SRAM cards are available for desktop

[1] Which is as of late 2010 in the rough ballpark of of $1–$3 per gigabyte for commodity SSD vs. $0.05–$0.50 per gigabyte for disk.

PCs, but they are specialty or enthusiast products and not typically available in the mass market; the least expensive of these are also limited to a fixed block device emulation and to the relatively low capacity of 4 GB [47]. DRAM-based storage is also being used in high end enterprise storage products, such as TMS' RamSan products [67] and Kaminario's DRAM storage appliance [38].

Chapter 3

Compression of Metadata

In this chapter we address the compressibility of metadata. First, we review the structure and baseline sizes of uncompressed Unix metadata (inodes) in Section 3.1. This is followed by an overview of our data collection methods and the systems we collected data from in Section 3.2. We compare mechanisms of inode compression, including three static compressors (all-or-nothing, precomputed Huffman table, and gamma compression) in Section 3.3. Finally, we present our the results of running those compressors on sample file systems in Section 3.4, showing that inodes are highly compressible using static compressors.

3.1 Baseline sizes

Most of the systems we analyzed used a version of the Unix file system semantics; 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) [48] and the Linux `ext2` file system [6], 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 individual file blocks or indirect blocks. The two newer related file systems, `ext4` and current versions of `ext3`, default to a larger, 256-byte inode; on `ext3` the extra space is entirely used for non-core metadata (fast symbolic links, extended attributes, and either extents or additional block pointers) while `ext4` extends the basic data beyond 128

bytes.[1] The file systems used by Windows (NTFS or FAT/FAT32) do not use inodes as the data structure for metadata storage, but the basic data recorded is largely similar on NTFS.

With the 128-byte inode used in the `ext2` file system as of the later 2.4 kernel versions of the Linux kernel, 74 bytes were used for block pointers and reserved free space; the remaining 54 bytes contain information that must be kept for each file.[2]

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 [74]. 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 as these would be kept on disk. If the indexes for these are kept in memory, compression techniques would be applicable to them as well—both block pointers (as in `ext2` or `ext3`) and extents (`ext4`) might be amenable to some form of compression. Alternatively, for large files, a single pointer to an on-disk structure could be maintained.

3.2 Data Collection

To study the compressibility of metadata, we first gathered data on current systems to serve as a sample on which to try different compression algorithms.

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 file 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

[1] As found via code inspection; in particular, several fields are now 64 bits—for example, to allow nanosecond values in the case of the timestamp fields.

[2] As of 2.6.35; this does not appear to have changed in practical terms for desktop use although several formerly reserved fields are now designated for special purpose features such as 32-bit UID/GID.

Table 3.1: File system profiles.

System	Total Files	Total ACLs	% of System	Total UIDs	Common Size
Redhat 8 root (2002)	213,569	119	98.8%	5	4–8 KB
Linux server root (2002)	431,615	165	59.5%	4	1–2 KB
Linux server `/home` (2002)	378,842	78	4.5%	4	64–128 KB
Workstation root (2010)	614,848	154	98.9%	19	4–8 KB
Linux server root (2010)	1,897,649	325	91.1%	45	4–8 KB
Linux server `/home` (2010)	1,444,090	160	1.3%	12	4–8 KB
UNIX server (all files)	1,618,855	10,417	28.8%	158	0.5–1 KB
Faculty directories	1,048,577	2,299	–	–	2–4 KB
Grads directories	1,141,004	7,554	–	–	128–256 B
SSRC shared dirs	789,376	1,229	–	–	1–2 KB

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 [58, 68].

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 [58, 68]. 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. We did some initial compression tests on these dumps, as described in Section 3.3 and Figure 3.1.

Based on these initial tests, we proceeded to gather additional traces of large multi-user file systems from our departmental file systems because the most "difficult" system to compress was the large UNIX server. We performed the remainder of the

analysis on the departmental file systems, the UNIX server file system, and representative file systems from a Linux workstation, as summarized in Table 3.1, which shows various characteristics of each of the seven file systems. Table 3.1 lists, for each file system, the number of files (active inodes with more than one link to them), the number of unique "ACLs" [3], the 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 class of files as grouped by powers of two.

This table also contains values for Linux systems from 2010, collected using the tools developed for the original paper; the three 2010 file systems come from a little-used laptop installation of Linux, and from the root and home file systems of the (several times upgraded) home server used in the 2002–2003 figures. While the number of files and amount of date have increased greatly, the basic profiles do not appear to have changed to a similar degree—the biggest difference is a moderate increase in the number of ACLs. Looking at the detailed dumps, on the two root file systems around 70% of all files share the same ACL (root:root, 644), and the frequency of other ACLs falls off quickly. On the Linux homes directories from 2010, the distribution looks increasingly like what was seen on the UNIX system from 2002: a handful of common permission sets multiplied across the set of users.

3.3 Inode Compression Mechanisms

We evaluated four different inode compression techniques. As a control, we used a conventional adaptive compressor, `deflate`, from the `zlib` compression library [27]. We tested this algorithm both on binary copies of individual inodes and on a single binary file containing the full set of inodes. The three remaining compressors were static compressors, tuned specifically for inodes.

The first static compression mechanism we evaluated for compressing inodes was a very simple *all or nothing* prefix compressor that encoded fields as shown in Table 3.2.

The second compression mechanism we evaluated for inode compression was the use of precomputed Huffman codes. These were based on the distribution of fre-

[3] In this case, we actually mean pseudo-access control lists—unique UID:GID:Mode triplets.

Table 3.2: 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.

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

quencies 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 with initially minimum frequency to indicate **OTHER**. This code would be followed by the full regular value for an `ext2` inode. For discrete value fields, it is used to represent values not known at the time the tree was generated.

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 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 [77, 24], shown in Table 3.3. 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 3.3: Sample table for gamma compression.

Unary	Length in bits	Minimum value	Range
0	1	0	0–1
10	3	2	2–9
110	7	10	10–137
1110	12	138	138–4233

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.

One additional refinement we used with gamma compression was to implement a pseudo-ACL mechanism. This system replaced the UID, GID, and Mode fields with references into a table containing UID/GID/Mode triplets. This reduced 48 bits[4] to a maximum of 14, and reduced the number of compression operations per inode by two, at the expense of the table lookup operations. It also, in theory, would allow for the easy replacement of the standard Unix user:group permission system with a more flexible ACL mechanism. Based on the data we collected, this should scale to systems with moderately large numbers of users and groups, but the practicality of scaling this to systems with very large numbers is unknown.

3.3.1 Implementation

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 [72]. We also modified the same scanner to compress the inodes

[4]The length of 48 bits as of 2002 has been increased to 80 bits with support for 32-bit UID and GID in newer versions of `ext2/3/4`.

with `zlib` using both the block compression and stream compression modes [27], 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.

3.4 Inode Compression Results

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 10 runs measuring to the nearest second, without writing any of the dump files to disk.

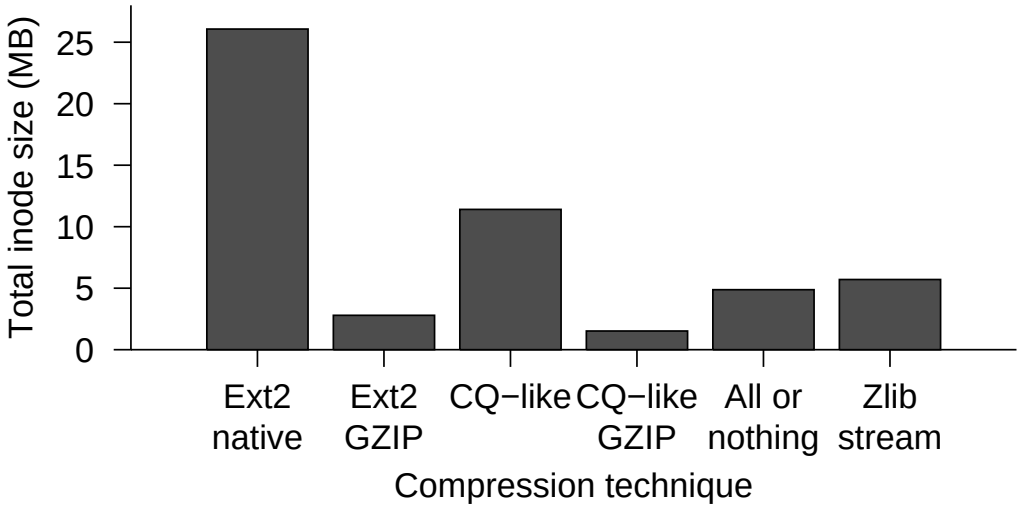

Figure 3.1: Initial compression results. "CQ-like" inodes are those stripped in a way similar to that in Conquest [74].

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 3.1. 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 achieved, it presents a reasonable 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 [27], 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 3.2, we first gathered more complete inode information on additional UNIX systems. We generated profiles—frequencies, gamma tables, and Huffman trees-l–for each of the seven file systems on which we ran tests, and then manually coded all-or-nothing compressors for each of the file systems. For each file system, we tested each compressor, using first using the profile produced from that file system, and then the other profiles from the other file systems. 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 seven cases when the profile matched the file system being compressed. In four cases out of seven, Huffman compression achieved the greatest space reduction. In the remaining three, gamma compression performed best. In all seven cases, all-or-nothing compression performed worse than either gamma or Huffman compression. It should also be noted that the difference between best and worst profiles was less significant for gamma compression than either Huffman or all-or-nothing.

As shown in Figure 3.2, best-case compressed inode sizes ranged from 14 to 19

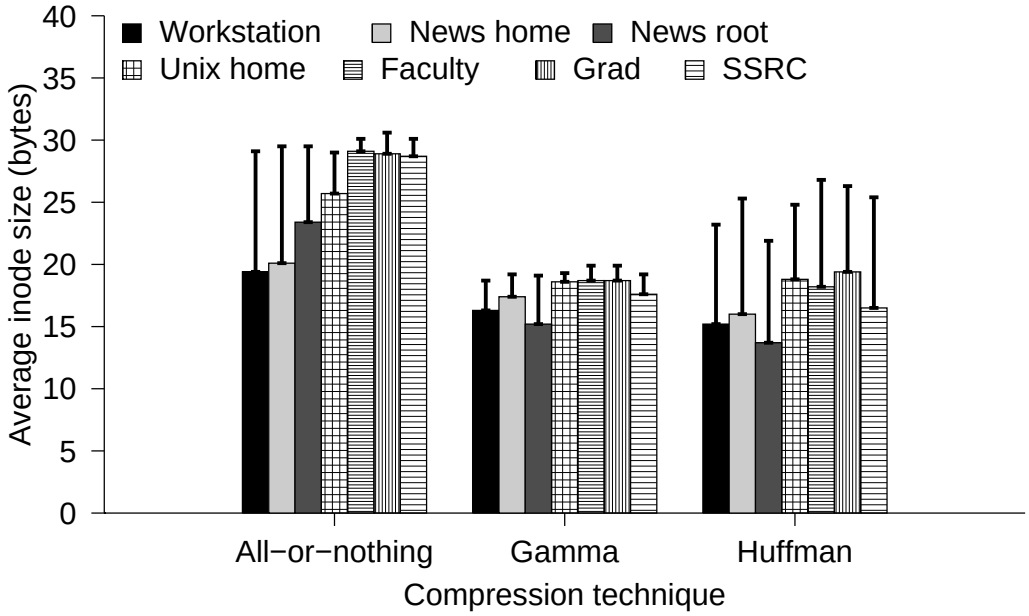

Figure 3.2: Average bytes per inode using the best and worst profiles for each file system. The colored bar for each compression technique shows the compressed size using the best profile, and the thin bar shows the range of compressed sizes using different file system profiles.

bytes, when using the best compressor and the profile generated from the original file system. 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. All compression tests were run on a 1.8 GHz Pentium 4 processor. These tests read the full set of inodes into memory and preallocated buffers for the compressed inodes before attempting any compression. The rates of compression for the gamma and Huffman overlapped slightly, with Huffman running at 270,000–600,000 inodes per second, and Gamma processing 480,000–600,000 inodes per second. The all-or-nothing compressor was somewhat faster, compressing 800,000–950,000 inodes per second. Decompressing inodes was significantly faster, achieving a rate of 2.2–2.7 million inodes per second.

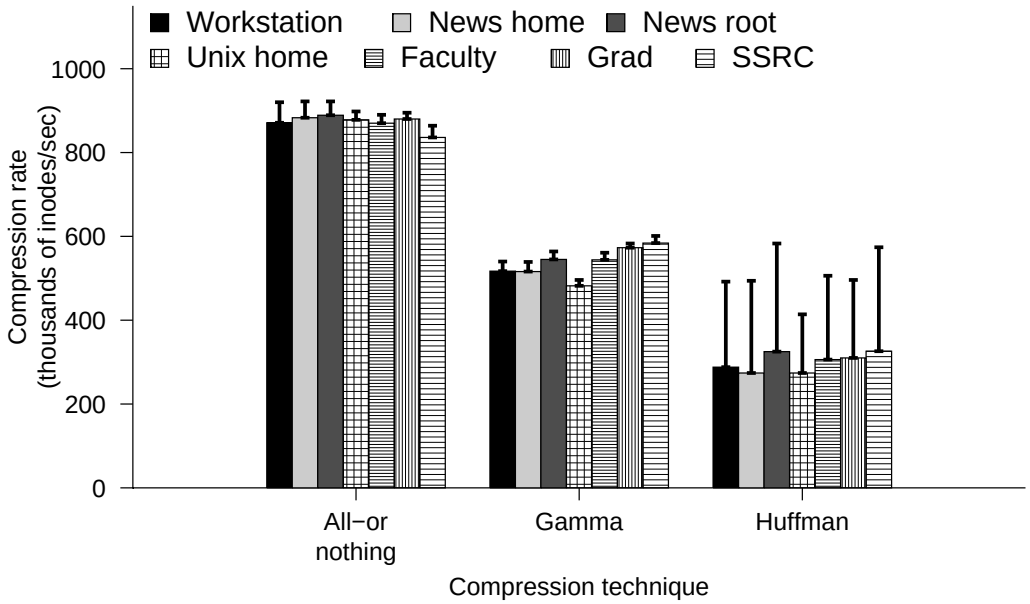

Figure 3.3: Compression rate, in thousands of inodes per second, for the best and worst profiles on each file system. The top of the shaded bar is the rate for the worst profile, and the top of the thin bar is the rate for the best profile.

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 3.4 compares uncompressed size with the measured best, average, and worst cases for the Huffman and gamma compressors averaged across all seven file systems.

The results for the timestamp values show the difficulty in compressing these values. The typical cases for both codes are still quite long, and in one case shows that a degenerate case may be *longer* than the standard 32-bit value. This already included several minor optimizations, including storing the creation time (CTime) as a delta from the millennium rather than the Unix epoch, and the modification time (MTime) and access time as deltas from the CTime and MTime respectively.

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

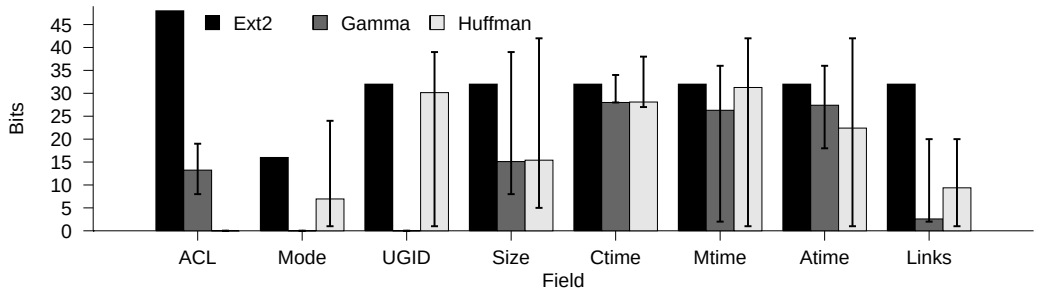

Figure 3.4: Compression effectiveness for different inode fields. For each field, the average field size is at the top of the shaded bar, and the error bars reflect the minimum and maximum field sizes. Note that gamma compression treats user IDs and mode bits as a single field; thus, it uses 0 bits for the individual fields.

compressed inodes, along the lines of the cleaner in a log-structured file system [8, 59], 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. One simple example of this is the pseudo-ACL mechanism we implemented. As noted by Reidel, *et al.* [56], the number of unique permission sets in a file system is relatively small, and, as shown in Section 3.2, on some file systems many files fall into the category of "system files" and could be represented by a small encoding in either Huffman or gamma compression.

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 ever actually used in production systems, and if so, what kind of distributions they fit. Similarly, all of the encoding methods allow 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.

Chapter 4

Compressing Small Files

In this chapter we review the motivation for compressing small files and discuss conventional methods of file data compression. We present our test setup comparing three different Lempel-Ziv variants across three different file systems. Finally, we show results of our tests indicating that small files are typically highly compressible with compression speeds not slower than that of desktop disk transfer rates.

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, large files will need be stored on disk, while it may be practical to store small files 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 standard 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 [81, 82], although one broadly used compression program, bzip2 [62], uses a block-sorting algorithm based on Burrows-Wheeler transforms [7].

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 neither the large UNIX server nor the departmental file server was available for these tests. The compression tests were also performed on an additional Linux workstation that 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 different adaptive compression algorithms. These were all block compressors of the Lempel-Ziv family. `Deflate` from the zlib library [27] is a relatively recent variant of LZ77 [81] 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) [52] and LZRW1 (Lempel-Ziv-Ross-Williams) [75]. 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 [16, 20, 76, 31].

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 increasingly 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 bins 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 overall. Files below 4 KB showed a decreased degree

Compressor	Average Compression	Average Rate	512 B–4 KB	16k B–128 KB
Deflate	61%	6.3 MB/sec	1–15 MB/s	26–38 MB/s
LZO	50%	36.8 MB/sec	1–13 MB/s	30–57 MB/s
LZRW1	44%	52.5 MB/sec	2–20 MB/s	35–54 MB/s

Table 4.1: Average file compression and speed range by compression technique and file size class.

of compressibility.

The graph we generated appeared to suggest that that the variability between individual size bins increases as we approach 128 KB although it has been suggested in online commentary about our original paper that this is an artifact of using bins with strictly equal file size ranges (varying the number of files per bin). The comment goes on to suggest the alternative of bins with an equal number of files (varying the range of file size per bin) [30].

Figure 4.1 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 4.2 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.

Figures 4.1 and 4.2 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 4.1. The figures in Table 4.1 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 com-

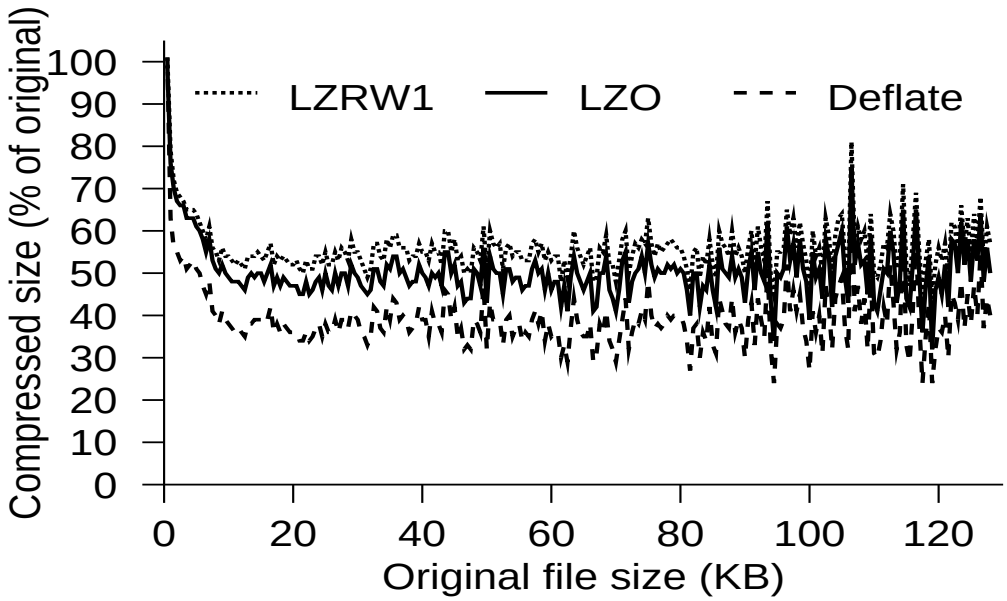

Figure 4.1: Compressibility of files on the root file system of the Linux workstation, calculated across a range of file sizes. The top line shows compression for LZRW1 is the top, and the bottom line for `deflate`.

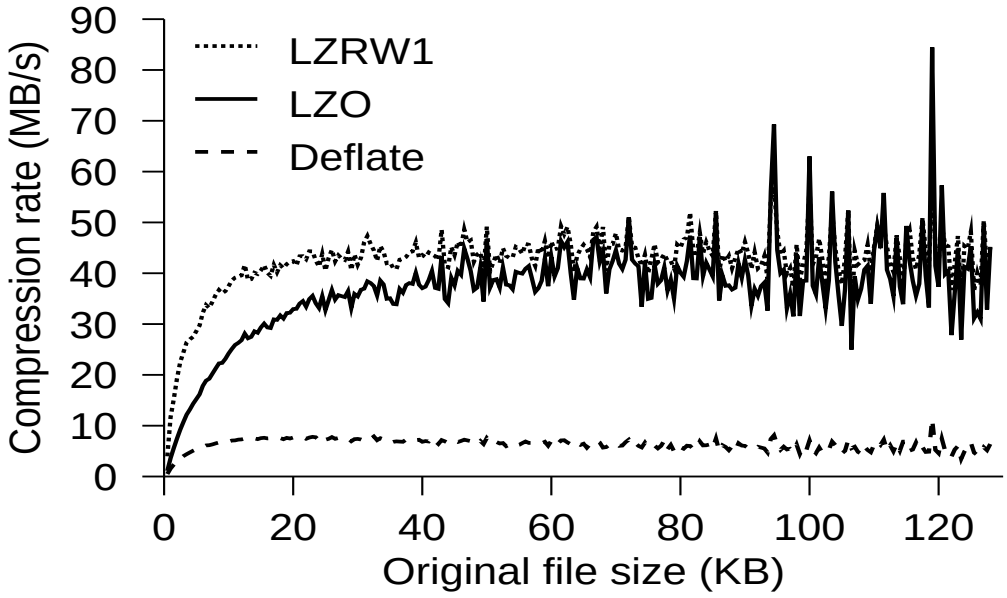

Figure 4.2: Speed of compression for the files on the root file system of the Linux workstation, calculated across a range of file sizes. The top line shows the speed for LZRW1, and the bottom line for `deflate`.

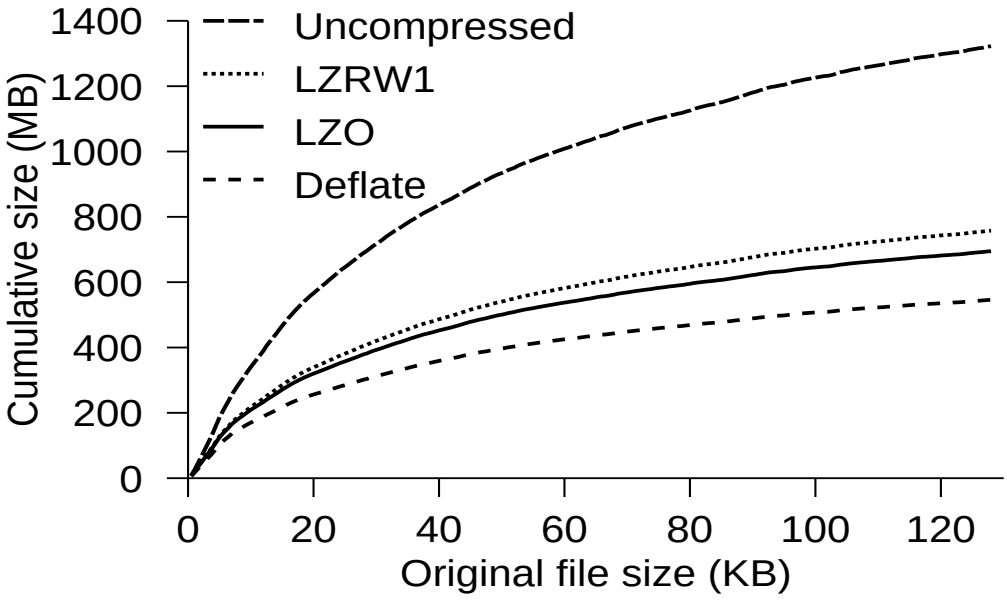

Figure 4.3: Cumulative space required for compressed files.

pressibility and rate of compression. On examination of the disk's contents, this was caused by the large number of JPEG images ranging from thumbnails (2–6 KB) to much larger files. Compressed file formats such as JPEG typically cannot 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`) or magic bytes at start of file data.

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 4.3. 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; at the time this work was originally done in 2002, the lowest figure of 570 MB remained 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 represented a substantial cost savings at 2003–2004 memory prices with the only drawback being the inclusion of file compression in the operating system. While the cost of memory has gone down substantially and typical size of main memory has increased, the size of operating systems and applications has increased as well—around 5.5 GB today for a full workstation installation of Linux (up from around 2 GB in 2002) with similar increased between then-current and present versions of Windows. The proportion of space taken by small files on a base Linux installation has increased a bit more slowly; files under 128 KB total about 2.3 GB out of the 5.5 GB vs. the example of a 1.3 GB out of about 2 GB.

Compressing files on NVRAM may have 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 time needed to read or write such files. A log-structured NVRAM file system such as JFFS2 [78] must pay an overhead to clean "segments;" the cleaning rate is proportional to the rate at which data is written to the file system [59]. By reducing the size of files via compression, we can reduce the overhead necessary to perform segment cleaning. Similarly, flash memory—and possibly other NVRAM technologies—degrade as blocks are erased and written repeatedly. Compression reduces the total amount of bytes written to the NVRAM, which may extend its lifetime, without reducing the amount of user data that can be stored on it.

These tests showed that the faster algorithms could keep up then-current desktop disk transfer rates (25–50 megabytes/second on a typical IDE drive of the era) 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 4.1 shows a more detailed summary of our results. The tests were run on a processor which was already fairly dated in 2002–2003[1] and on a mainstream processor today, we expect that even `deflate` should be able to keep up with desktop disk transfer rates without fully loading the CPU.

[1]Specifically, an AMD Athlon XP 1700+ running at 1.1 GHz, comparable to some present netbook processors.

These tests, however, were operating on complete files of up to 128 KB 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, which reached full speed at file sizes of about 20 KB, while `deflate` reached full speed at about 8 KB. Similarly, all compressors achieved their overall average compression for files of about 12–16 KB and above; smaller files showed lesser compressibility.

There are other compression techniques which might be favorable for the smallest files. These include some of the block compressors developed and evaluated by Wilson, *et al.* [76] or compressors which use pre-existing knowledge about the data, such as canonical Huffman trees for English/ASCII text or using a pre-populated dictionary [77].

Chapter 5

The *mramfs* File System

In this chapter we introduce the file system *mramfs* which we created to test the metadata and file compression techniques within a live system. We first present the design of the system in Section 5.1. We then discuss the process of testing and benchmarking the prototype system and present the results of those benchmarks in Section 5.2.

5.1 Design and Implementation

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 metadata and data. This is done to model the case where the file system is stored in a persistent NVRAM buffer and rather than in main memory, and to simulate the use of slower memory. We simulate persistence, and use a large block of volatile ram. Persistence is implemented by copying the memory region to disk on unmount, and restoring it from disk when remounted. Unmounting 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 5.1, 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

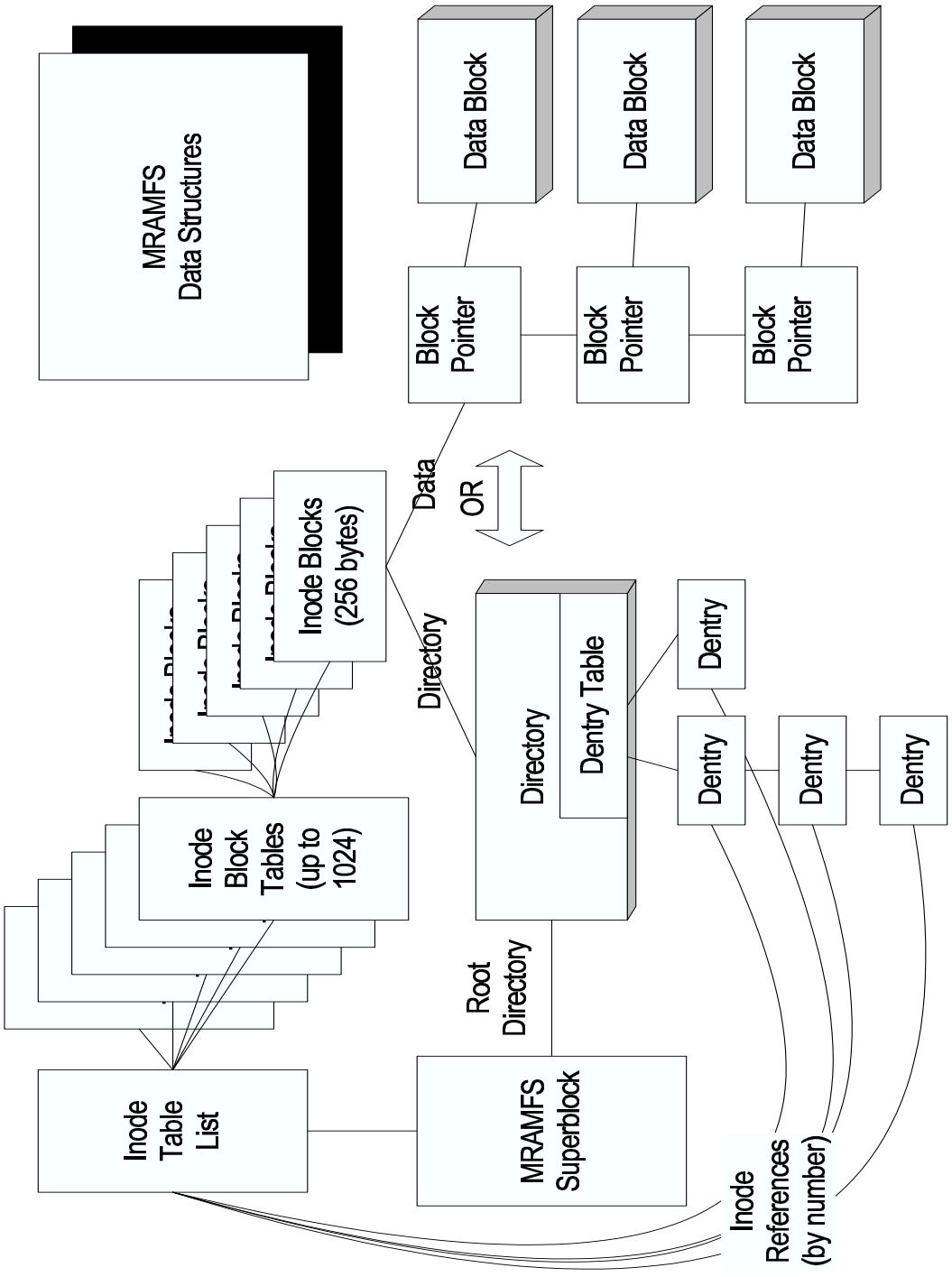

Figure 5.1: *mramfs* data structures.

superblock object, an empty root directory, and various memory management structures. One key difference is that all pointers are relative to the start of our memory region, to allow the image to be useable without having a fixed memory allocation across boots or even between different systems or kernel versions. In practice, this was never tested rigorously; the process of trying to get our code updated to compile on modern (2.6.30+) kernels and to run on both 32- and 64-bit kernels has revealed a few cases where this was missed and needed to be fixed.

Our file system utilizes a single large region of memory which is allocated using either *vmalloc* or an IO memory mapping. It is mapped into kernel space when mounted. This could trivially be adapted to use a directly mapped non-volatile memory if we had one available. With a little more efford, this could be readily adapted to either an indirectly mapped NVRAM where a only a partial window was mapped into the address space at any one time.

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. Recovering empty segments was never implemented; this will need to be implemented. It is also possible that an entirely different allocation scheme would perform better.

Directories are implemented as chained hash tables using a Jenkins hash [35], 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; this is not optimally space efficient. 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; it is likely possible to improve on this by using a linked list implementation for very small directories, and/or by rehashing to increasingly larger tables as the directory size increases. Given the very good performance of `ext2` on 100 subdirectories—see Section 5.2—an optimized path for small subdirectories would seem particularly promising.

Inodes are implemented to be either compressible or stripped, and packed into

```
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 5.2: Pseudocode for *mramfs* inode structure.

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 approximately 16 million inodes (2^{24}) 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 up to 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 5.2. 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 rewritten 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. The present implementation uses a fixed 32-bit value; this could be scaled to a smaller fixed number of bits based on the file system size and might also be scalable to a known level of alignment.

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 an 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. 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, uncompressed data blocks are all 4 KB, corresponding to the page cache size on a standard Linux system. Allowing larger date blocks (or better still, flexible data block sizes to store files contiguously) would likely significantly improve the degree of compression and reduce the overhead of compression headers and block pointers. 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. 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.

5.2 Benchmark and 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 then-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 `ext2` on RAMdisk. Creating and unlinking 4,000 files 4,000 times, for a total of 32 million metadata operations, took approximately 36 ± 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 then `ext2` on the Linux RAMdisk driver (`rd`). One alternative, the Linux Memory Technology Driver subsystem (`MTD`), primarily supports various forms of flash. 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 [78].

5.2.1 Postmark benchmark results

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 then current Linux kernel, on a faster system;[1] all tests 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 `ext2`, 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,

[1] 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.

as `ramfs` and `tmpfs` do not have a data representation separate from the Linux page cache. This makes them unsuitable for most forms of non-volatile memory, as it is unlikely that the entire main memory of a system will be non-volatile. 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 copying data by making the entire main memory non-volatile, this is not a case we find practical in the near future[2]. It also adds substantial further complexity to the operating system boot process itself as in Conquest [74] or requires non-PC-compatible hardware, as in the DEC Alpha systems used for the RIO File Cache [9].

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 and `BPFS` [11]. 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 easily 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. As showed by Conquest, this would be a definite gain as long as NVRAM operated at main memory speed (or faster); if it did not, as the difference in memory speeds increased, it would potentially become disadvantageous. However, it is much less practical for compressed data, as random reads are more difficult, and in place random writes essentially impossible.

One solution, with small compressed files, would be to handle compression for the entire file, rather than operating on individual blocks. Files being read could simply be decompressed completely into pre-allocated pages in main memory. Writes would require some additional complexity. Options for handling writes include a non-volatile write buffer or journal to store uncompressed writes until the file could be read and recompressed, storing writes as deltas (as in `JFFS2` [78], and potentially requiring cleaning) or doing a full file read before any in place writes.

Our results for Postmark are shown in Figure 5.3; these are results for several

[2]This applies to PC-class systems; embedded systems may be another matter.

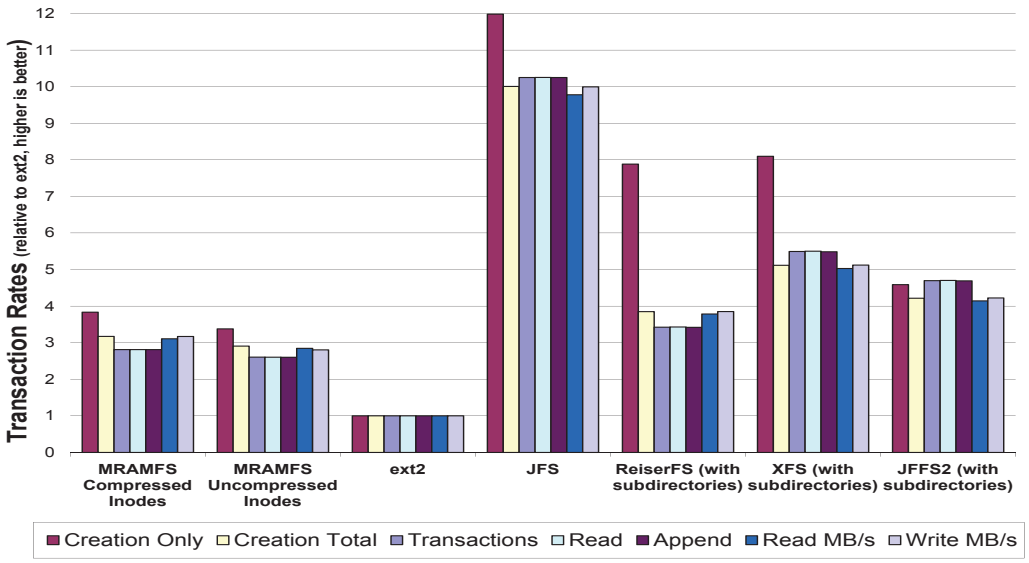

Figure 5.3: Postmark transaction rates for 100,000 transactions, 50,000 files. Unless noted otherwise, the benchmark was run against a single directory. The vertical axis represents transaction rates normalized to `ext2/3/4`; higher is better, representing a larger number of transactions per time period.

file systems, relative to `ext2` on RAM disk.[3] 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* and except where otherwise noted, in a single directory. The raw data for our final set of postmark runs is included in Appendix C.

We took `ext2` (running on RAM disk), as a baseline for comparison. Results for `tmpfs` and `ramfs` are not shown in Figure 5.3. 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 clearly supports our belief that gamma compression for inodes is a nearly free space savings.

We believe performance of `ext2` is the result of the extreme inefficiency of

[3]In this case, the RAM disk was the `MTD` block drive and not the older `/dev/ram` driver.

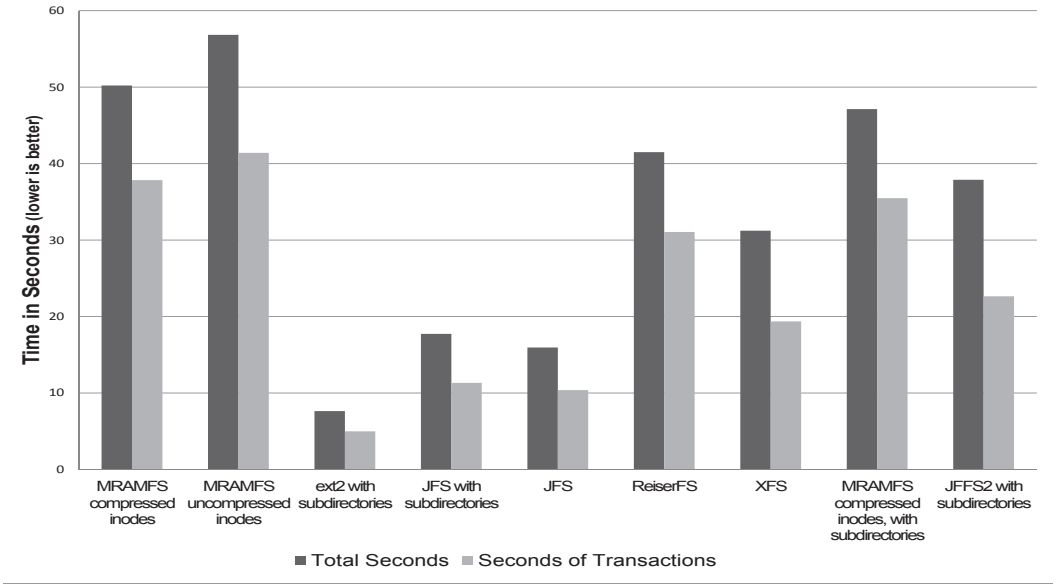

Figure 5.4: Postmark running times for 100,000 transactions, 50,000 files. Unless noted otherwise, the benchmark was run against a single directory. The vertical axis represents running times in seconds; lower is better.

linear-search data structures when dealing with very large numbers of files. This was true for `JFFS2` when *run in a single directory* as well, to a much greater degree—running Postmark took significantly in excess of 10 minutes, and was cancelled before completion.

We resolved this problem by rerunning the full set of tests with the Postmark option *100 subdirectories*. With the exceptions of `JFFS2` and `ext2`, all file systems performed comparably with either—no more than slightly better or worse in any case. Both `JFFS2` and `ext2` performed significantly better with smaller directory sizes; `JFFS2` was roughly comparable to the other file systems tested. With 100 subdirectories, `ext2` 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.

5.2.2 Build benchmark results

The results from our build benchmark are shown in Figure 5.5. We compare the build times (both total and system time) normalized relative to `tmpfs`. We compared the

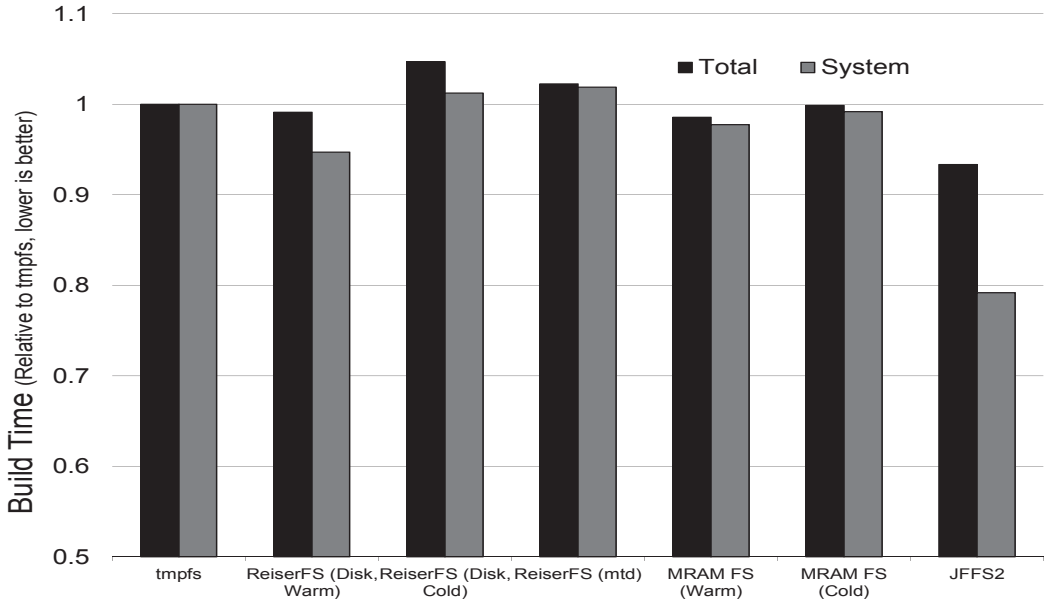

Figure 5.5: Build benchmark results.

full set of file systems from the Postmark benchmark, beyond those displayed, 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.

5.2.3 Compressibility

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 was never reliable enough to complete significant runs of Postmark or of large builds, so our preliminary performance analysis is based on very small tests. At the point work was terminated, it was performing unacceptably, at around 20–25% of the speed of regular *mramfs*. This may be a result of implementation flaws; compression speeds demonstrated by recent versions of the Linux compressed cache project are much better. [31]

Chapter 6

Future Work and Retrospective

In this chapter, we first discuss lessons learned and further work we had intended to complete related to the specific implementation of *mramfs* in Section 6.1. We follow this with a discussion of more general areas for consideration, either as future work or which have been addressed by more recent publications in Section 6.2.

6.1 *mramfs* in Retrospect

The file system we developed was a prototype/proof-of-concept; there are a number of factors which should be addressed, either as a matter of future research, or in order to convert this into a practical tool for production.

First, there are minor functions which needed to be implemented for a production file system; these include handling device nodes and other special files, implementing now-standard features such as Posix ACLs and extended attributes, and making sure that directory behavior was fully correct (this last was a siginifcant source of bugs in testing and benchmarking, especially as regards symlinks.)

It is also likely that moving to any particular physical NVRAM technology may require some additional support, although our implementation attempted to segregate all memory read/write access to a limited subset of utility functions. Of course, this prevents the use of execute-in-place and directly memory mapping data files which may be desirable memory is directly mappable and roughly as fast as main memory.

Second, the prototype is currently outdated; there have been several major

changes to the VFS layer between 2.6.7 and the present 2.6.37. Additionally, development was done on a single-threaded 32-bit CPU, and only a minimal effort made at the to make it runnable on a 64-bit or SMP system. This was an acceptable first step in 2003; it would not be today. Both of these issues would need to be addressed to make it a usable tool for further research today, let alone to be useful in any kind of production setting.

One other sense in which *mramfs* is outdated is in the implementation and debugging process we used; in 2003, user-space file system support on Linux was very much a moving target. It was not clear whether LUFS or FUSE was the better bet, and neither was well-documented compared to the VFS itself. We initially did some development on LUFS, but that had enough problems at the time that we abadoned it to move our implementation into the kernel relatively early on. FUSE proved a much more stable and rapidly evolving project, and would be the obvious choice today for stabilizing the system before moving it into the kernel.

We had intended to examine ways to make our directory structures more space efficient; some of these methods are obvious and can be expected to be computationally cheap, like using variable-length fields. Others, like hashing strings for deduplication, or using a fixed text compressor, would require further experiment to find out if the savings are significant and if the potential implementation is too computationally expensive.

The Jenkins hash function we used for file names was selected for its expedient availability and GPL-compatible license [35]; its appropriateness and performance could use validation. The use of hash functions (generally in hashed trees) in existing file systems like `ext3` and `ext4` may be instructive, although the demands of a purely in-memory hash table and an on-disk structure are likely to be different.

We had intended to rework file data compression, as the initial results failed to achieve satisfactory performance comparable to our userspace simulations. Our initial belief had been that this was a result of doing the compression on individual blocks, and that any implementation of that policy was going to be inherently slower. One alternative we had intended to explore was the comparison with full-file compression policies; this might also allow for data compression while moving from the current page-cache based read/write implementation to one using the file methods.

However, the performance achieved by the compressed cache project [31] suggests both that some or all of our performance problem may be the result of implementation details, and also provides a possible source of a better implementation of in-kernel block compression. If block-by-block compression is used, techniques for packing compressed blocks in a space efficient manner avoiding internal fragmentation bears further examination; Huang, *et al.* discuss this in the context of their compressing FTL [33].

Compression aside, we had intended to support variable size allocation in order to increase space efficiency for file data, both in terms in decreased internal fragmentation and a decreased need for block pointers. As an alternative, this could be handled by the use of extents with very small data blocks, as explored in `NEBFS` [61].

6.2 Future Work

The original intent had been to use this module as a tool in future research for file systems for non-volatile RAM. While it has been left as is, there are a number of broader areas we had hoped to examine; these include policies and mechanisms for splitting data between NVRAM and disk in a hybrid file system, space efficiency, and the performance impact of varying memory technologies. Some of these have been examined by more recent works.

In the area of metadata compression techniques, there are a number of possible avenues that can still be explored. One is the efficient encoding of time values, which tend to be fairly long bit strings if encoded individually. Additionally, while all of our tests up to now 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. This would also apply to avoiding attempts to recompress files containing compressed data; both our own file system profiling and existing studies [25] show the large numbers of compressed files, especially multimedia. Attempting to recompress these have an negative impact on compression speed and ratio. Being

able to recognize these files - or at least a common subset of these - without first trying to compress them would improve performance.

The use of multiple compression profiles on a single system, either for different file systems/mount points or at the inode or directory level, could yield higher compression rates. For one possible example, on small Linux or other UNIX-like systems, one could flag files as either "system" or "user" profiles. This could be further refined with adaptive techniques, 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 is compressible. One possible way to do this would be to use a fast adaptive block or stream compressor on groups of inodes-like structures on disk; this might on the one hand eliminate the high cost of a block header per individual file while maintaining relatively low-cost random access to any file's block pointers.

One issue we identified for future work but did not get to examine is reliability: possible 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. This issue has been examined in in much greater depth in recent works, including those by Greenan [29] and Kang [39]; another example is the short-circuit shadow paging technique suggested by the `BPFS` system [11].

One area we did not identify at the time for future work, but which has become increasingly important is power efficiency. Work in this area has focussed both on limiting mechanical disk access on PC and server class machines, for example work on smart spindown [3, 4], and also on modeling the power efficiency of flash file systems for embedded or mobile systems [10, 28].

While *mramfs* is a tradition UNIX/FFS-like file system adapted to memory, it would be interesting to see how the in-memory compression techniques applied to a one of the more recent tree-based file systems; the design of `BPFS` [11] provides one basis for how such a system might be applied to NVRAM, and some early reports of very good compression performance with similar on-disk file systems like `zfs` [44] and `btrfs` [43]

suggests that this should receive further examination.

Lastly, one potential application for these techniques would be to examine whether there is any potential performance benefit to using them to increase the amount of purely volatile memory available for metadata caches. While this seems unlikely with typical PC workloads on relatively memory-rich hardware, it might be well suited to specific metadata-heavy workloads or to relatively memory-constrained mobile or embedded hardware, although in those cases a possible tradeoff on CPU load and power efficiency would need to be addressed.

Chapter 7

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 metadata and file data are highly compressible with little increase in code complexity. 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.

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. Our file system performed slightly better on the modified Postmark benchmark when compression is enabled as compared to disable, and the slowest compressor we evaluated averaged less than four microseconds per inode, an improvement of 250:1 over a 1 millisecond disk access. The fastest compressors we evaluated were 3–4 times faster still. Finally, even as compared to purely in-memory file systems, compression offers very close performance for metadata operations.

Similarly, compressing small files will improve performance by increasing how many small, latency-sensitive files which can be kept in NVRAM for a given capacity;

our initial simulation tests showed doubling capacity is readily achievable. On then-current processors the average compression rates for LZRW1 and LZO could match the typical data rates of typical desktop disk systems. At present, the balance has moved further in favor of CPUs. 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. While our prototype implementation of data block compression was not successful, we believe that this was problem of implementation, and that a more refined implementation would achieve significantly better performance. This is also suggested by the promising performance figures for the compressed swap driver for Linux [31].

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.

Appendix A

Code Availability

Source code for the *mramfs* module is distributable under the GPL. Versions were created for 2.4 and early 2.6 kernels; it has not been updated for later versions of 2.6.

The final 2.6 version which is believed to compile on versions up to 2.6.7. It is available for download at `https://github.com/nkedel/MRAMFS`

Appendix B

Design notes

```
Memory allocation:
* Large buffer of 64MB to 2GB (easily extendable to 16GB)
* Buffer is divided into 64KB segments; segment 0 is reserved,
  remainder are allocated by bitmap
* Segments are split into like-sized small objects (16B - 32KB)
* Small objects are stored in free lists by superblock

Major data structures:
----------------------
mramfs_super:
* kept in kernel memory and mirrored in memory buffer by
  read_super/put_super
* keeps private information:
    table of inode tables,
    root directory pointer,
    memory allocation free lists,
    segment allocator master pointers,
    pointers to acl_hash and acl_table
    file and block counts

acl_table: ordered table of acls
           acl table entry contains UID, GID, permissions

acl_hash: hash pointers to acls

directory objects:
    hashtable of dentry objects
    file count
    pointers to owning and parent inodes
```

```
dentry object: name, inode number (plus next value in list for hash table)

Inodes
------
Three level structure:

Table of tables (mramfs_super->inodes) --
    table of 1024 inode tables, allocated at FS creation time
Inode block tables --
    table of 1024 inode block pointers, allocated on first use
    (never deallocated)
Inode blocks --
    256-byte block of packed inodes, allocated on first use
    (never deallocated)

Inode block format, uncompressed:
    Maximum 4 million inodes --
        block_offset: inode_num \& 3 (bits 0-1)
        block_number: (inode_num >> 2) \& 1023  (bits 2-11)
        table_number: (inode_num >> 12) \& 1024 (bits 12-21)
    Bytes 0..1: 16 bit "inodes in use" bit-mask, bits 0..3 used
    Bytes 2..255: Stores 4x ~50-byte uncompressed inodes
         (plus wasted space)
    Uncompressed inodes stored at fixed offsets (2, 64, 128, 192);
    to find a given inode, just go to offset within block.

Inode block format, compressed:
    Gamma compressed fields, plus pseudo-ACL mechanism
    Bytes 0..1: 16 bit "inodes in use" bit-mask, all bits
    Bytes 2..255: Stores UP TO 16x 15-63-byte compressed inodes
    Maximum 16 million inodes --
        block_offset: inode_num \& 15 (bits 0-3)
        block_number: (inode_num >> 4) \& 1023  (bits 4-13)
        table_number: (inode_num >> 14) \& 1024 (bits 14-23)
    Compressed inodes are stored packed, in the form:
        Byte 0: len
        Bytes 1..len: compressed data (padded to byte boundary)
        Byte len+1:   beginning (len) of next packed inode.
    To find a given inode n:
     traverse n-1 prior inodes as a linked list.

Since no compressed inode can be as short as 0 bytes, we mark a deleted
```

```
 inode by a length of 0 (an empty inode block is simply 18 zero bytes.)

Problem: recompressing an inode tends to make it bigger.
Current solution:
* If there are sufficient slack bytes at the end, move the remaining
  bytes up.
* If there aren't sufficient slack bytes at the end, we relocate the
  inode data to a new inode.

Possibly also: * preallocate 18 bytes rather than 15
    (this would allow at most 14 inodes per block; similarly could
     preallocate 19, for at most 13 inodes/block)

Since no compressed inode can be as short as 4 bytes, we mark a relocation
 by any inode with a length of 4.

Problem: we relocate very frequently in a full file system.
Solution:
* As inode blocks start to fill up, switch those offsets within the
  block that are marked as empty to marked as used.
* Never allow double relocations;
    if we're replacing an inode that's been relocated, and the
    new inode won't fit, delete the relocation and relocate to an
    entirely new inode
* Try to move back relocations:
    If we can fit an entire inode that was previously relocated,
    delete the previous relocation.

Finding a free inode:
  scan starting at a given inode table^
  if inode table is NULL, first inode in table is free
    if inode table is not NULL, scan over all blocks in table
      if block is NULL, first inode in block is free
      if block is not NULL, check mask:
         if mask is all set (-1 short/0xFFFF) block is full
         otherwise free space in block, scan over mask to find 0 bit
         -- bit number of free bit indicates

^ in particular, for relocations, we never want to relocate to a
  lower inode number
```

Appendix C

Postmark benchmark data

```
MRAMFS (compressed inode, uncompressed blocks)
50,000 files/100,000 transactions/1-4095 bytes
Run1:
        50.423000 seconds total
        38.110000 seconds of transactions (2623.983207 per second)

Files:
        99928 created (1981.794023 per second)
                Creation alone: 50000 files (4100.713524 per second)
                Mixed with transactions: 49928 files (1310.102335 per
second)
        49907 read (1309.551299 per second)
        49997 appended (1311.912884 per second)
        99928 deleted (1981.794023 per second)
                Deletion alone: 49856 files (415466.666667 per second)
                Mixed with transactions: 50072 files (1313.880871 per
second)

Data:
        112.94 megabytes read (2.24 megabytes per second)
        235.99 megabytes written (4.68 megabytes per second)

Run2:
Time:
        50.215000 seconds total
        37.839000 seconds of transactions (2642.775972 per second)

Files:
        99731 created (1986.079857 per second)
```

```
                Creation alone: 50000 files (4080.300310 per second)
                Mixed with transactions: 49731 files (1314.278919 per
second)
        49905 read (1318.877349 per second)
        50009 appended (1321.625836 per second)
        99731 deleted (1986.079857 per second)
                Deletion alone: 49462 files (405426.229508 per second)
                Mixed with transactions: 50269 files (1328.497053 per
second)

Data:
        112.91 megabytes read (2.25 megabytes per second)
        235.60 megabytes written (4.69 megabytes per second)

MRAMFS (uncompressed inode, uncompressed blocks)
50,000 files/100,000 transactions/1-4095 bytes
Run1:
Time:
        54.913000 seconds total
        40.848000 seconds of transactions (2448.100274 per second)

Files:
        99928 created (1819.751243 per second)
                Creation alone: 50000 files (3594.794737 per second)
                Mixed with transactions: 49928 files (1222.287505 per
second)
        49907 read (1221.773404 per second)
        49997 appended (1223.976694 per second)
        99928 deleted (1819.751243 per second)
                Deletion alone: 49856 files (319589.743590 per second)
                Mixed with transactions: 50072 files (1225.812769 per
second)

Data:
        112.94 megabytes read (2.06 megabytes per second)
        235.99 megabytes written (4.30 megabytes per second)
Run2:
Time:
        56.837000 seconds total
        41.405000 seconds of transactions (2415.167250 per second)

Files:
        99928 created (1758.150501 per second)
```

```
                Creation alone: 50000 files (3281.916639 per second)
                Mixed with transactions: 49928 files (1205.844705 per
second)
        49907 read (1205.337520 per second)
        49997 appended (1207.511170 per second)
        99928 deleted (1758.150501 per second)
                Deletion alone: 49856 files (253076.142132 per second)
                Mixed with transactions: 50072 files (1209.322546 per
second)

Data:
        112.94 megabytes read (1.99 megabytes per second)
        235.99 megabytes written (4.15 megabytes per second)

ext2 (mtdblock)
50,000 files/100,000 transactions/1-4095 bytes
Time:
        159.565000 seconds total
        106.300000 seconds of transactions (940.733772 per second)

Files:
        99928 created (626.252624 per second)
                Creation alone: 50000 files (1064.169416 per second)
                Mixed with transactions: 49928 files (469.689558 per second)
        49907 read (469.492004 per second)
        49997 appended (470.338664 per second)
        99928 deleted (626.252624 per second)
                Deletion alone: 49856 files (7938.853503 per second)
                Mixed with transactions: 50072 files (471.044214 per second)

Data:
        112.94 megabytes read (724.81 kilobytes per second)
        235.99 megabytes written (1.48 megabytes per second)

ext2 (mtdblock) w/ subdirs
50,000 files/100,000 transactions/1-4095 bytes
Time:
        7.628000 seconds total
        4.990000 seconds of transactions (20040.080160 per second)

Files:
        100001 created (13109.727320 per second)
                Creation alone: 50000 files (24789.291026 per second)
```

```
                Mixed with transactions: 50001 files (10020.240481 per
second)
        49966 read (10013.226453 per second)
        49927 appended (10005.410822 per second)
        100001 deleted (13109.727320 per second)
                Deletion alone: 50002 files (80518.518519 per second)
                Mixed with transactions: 49999 files (10019.839679 per
second)

Data:
        113.67 megabytes read (14.90 megabytes per second)
        236.57 megabytes written (31.01 megabytes per second)

jfs (mtdblock) - subdirs
50,000 files/100,000 transactions/1-4095 bytes
Time:
        17.735000 seconds total
        11.323000 seconds of transactions (8831.581736 per second)

Files:
        100001 created (5638.624189 per second)
                Creation alone: 50000 files (12297.097885 per second)
                Mixed with transactions: 50001 files (4415.879184 per
second)
        49966 read (4412.788130 per second)
        49927 appended (4409.343813 per second)
        100001 deleted (5638.624189 per second)
                Deletion alone: 50002 files (21313.725490 per second)
                Mixed with transactions: 49999 files (4415.702552 per
second)

Data:
        113.67 megabytes read (6.41 megabytes per second)
        236.57 megabytes written (13.34 megabytes per second)

jfs (mtdblock) - no subdirs
50,000 files/100,000 transactions/1-4095 bytes
Time:
        15.953000 seconds total
        10.379000 seconds of transactions (9634.839580 per second)

Files:
        99928 created (6263.900207 per second)
```

```
                Creation alone: 50000 files (12748.597654 per second)
                Mixed with transactions: 49928 files (4810.482705 per
second)
        49907 read (4808.459389 per second)
        49997 appended (4817.130745 per second)
        99928 deleted (6263.900207 per second)
                Deletion alone: 49856 files (30179.176755 per second)
                Mixed with transactions: 50072 files (4824.356874 per
second)

Data:
        112.94 megabytes read (7.08 megabytes per second)
        235.99 megabytes written (14.79 megabytes per second)

reiserfs (mtd) - subdirs
50,000 files/100,000 transactions/1-4095 bytes
Time:
        41.499000 seconds total
        31.061000 seconds of transactions (3219.471363 per second)

Files:
        100001 created (2409.720716 per second)
                Creation alone: 50000 files (8386.447501 per second)
                Mixed with transactions: 50001 files (1609.767876 per
second)
        49966 read (1608.641061 per second)
        49927 appended (1607.385467 per second)
        100001 deleted (2409.720716 per second)
                Deletion alone: 50002 files (11171.134942 per second)
                Mixed with transactions: 49999 files (1609.703487 per
second)

Data:
        113.67 megabytes read (2.74 megabytes per second)
        236.57 megabytes written (5.70 megabytes per second)

xfs-mtd-subdirs
50,000 files/100,000 transactions/1-4095 bytes

Time:
        31.223000 seconds total
        19.366000 seconds of transactions (5163.688939 per second)
```

```
Files:
        100001 created (3202.799219 per second)
                Creation alone: 50000 files (8614.748449 per second)
                Mixed with transactions: 50001 files (2581.896107 per
second)
        49966 read (2580.088815 per second)
        49927 appended (2578.074977 per second)
        100001 deleted (3202.799219 per second)
                Deletion alone: 50002 files (8260.697175 per second)
                Mixed with transactions: 49999 files (2581.792833 per
second)

Data:
        113.67 megabytes read (3.64 megabytes per second)
        236.57 megabytes written (7.58 megabytes per second)


mramfs compressed - subdirs
50,000 files/100,000 transactions/1-4095 bytes
Deleting subdirectories...Done
Time:
        47.128000 seconds total
        35.482000 seconds of transactions (2818.330421 per second)

Files:
        100001 created (2121.902054 per second)
                Creation alone: 50000 files (4345.558839 per second)
                Mixed with transactions: 50001 files (1409.193394 per
second)
        49966 read (1408.206978 per second)
        49927 appended (1407.107829 per second)
        100001 deleted (2121.902054 per second)
                Deletion alone: 50002 files (357157.142857 per second)
                Mixed with transactions: 49999 files (1409.137027 per
second)

Data:
        113.67 megabytes read (2.41 megabytes per second)
        236.57 megabytes written (5.02 megabytes per second)

JFFS2 (subdirectories)
50,000 files/100,000 transactions/1-4095 bytes
```

```
Time:
        37.880000 seconds total
        22.651000 seconds of transactions (4414.816123 per second)

Files:
        100001 created (2639.941922 per second)
                Creation alone: 50000 files (4880.905896 per second)
                Mixed with transactions: 50001 files (2207.452210 per
second)
        49966 read (2205.907024 per second)
        49927 appended (2204.185246 per second)
        100001 deleted (2639.941922 per second)
                Deletion alone: 50002 files (10030.491474 per second)
                Mixed with transactions: 49999 files (2207.363913 per
second)

Data:
        113.67 megabytes read (3.00 megabytes per second)
        236.57 megabytes written (6.25 megabytes per second)

tmpfs
50,000 files/100,000 transactions/1-4095 bytes
Time:
        2.882000 seconds total
        1.962000 seconds of transactions (50968.399592 per second)

Files:
        99928 created (34673.143650 per second)
                Creation alone: 50000 files (65104.166667 per second)
                Mixed with transactions: 49928 files (25447.502548 per
second)
        49907 read (25436.799185 per second)
        49997 appended (25482.670744 per second)
        99928 deleted (34673.143650 per second)
                Deletion alone: 49856 files (328000.000000 per second)
                Mixed with transactions: 50072 files (25520.897044 per
second)

Data:
        112.94 megabytes read (39.19 megabytes per second)
        235.99 megabytes written (81.88 megabytes per second)

Build benchmark results:
```

```
Disk - ReiserFS - Warm Cache/Tails
real    2m5.348s
user    1m13.733s
sys     0m49.267s

Disk - ReiserFS - Cold Cache/Tails
real    2m12.421s
user    1m14.264s
sys     0m52.669s


MRAMFS - Compressed Inodes - Warm Cache
real    2m4.648s
user    1m13.588s
sys     0m50.856s

MRAMFS - Compressed Inodes - Cold Cache
real    2m6.290s
user    1m14.231s
sys     0m51.600s

OVERALL   GAvg      GMax      GMin
    2915      122      145      113
Field GAvg     GMax     GMin
ACL          5        5        5
Size        11       34        2
Ctime       31       31       31
Mtime       18       18       18
Atime       18       18       18
Links        2        6        2
Vers.        2        2        2

Disk - Ext2 - Warm Cache
real    2m8.311s
user    1m14.451s
sys     0m51.696s

Disk - Ext2 - Cold Cache
real    2m9.086s
user    1m13.952s
sys     0m50.983s
```

```
tmpfs
real    2m6.453s
user    1m14.398s
sys     0m52.021s

jfs - mtd
real    2m8.947s
user    1m13.901s
sys     0m51.596s

reiserfs - mtd
real    2m9.292s
user    1m13.817s
sys     0m53.009s

jffs2 - mtd
real    1m58.055s
user    1m13.748s
sys     0m41.191s
```

Bibliography

[1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo 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, October 1992.

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

[3] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. Nvcache: Increasing the effectiveness of disk spin-down algorithms with caching. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 422–432, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. A hybrid disk-aware spin-down algorithm with i/o subsystem support. In *Proceedings of the International Performance Conference on Computers and Communication (IPCCC '07)*, 2007.

[5] Hans Boeve, Cristophe Bruynseraede, Jo Das, Kristof Dessein, Gustaaf Borghs, Jo De Boeck, Ricardo C. Sousa, Luís V. Melo, and Paulo 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, September 1999.

[6] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, 2nd edition, December 2002.

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

[8] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy 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, October 1992.

[9] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David 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, October 1996.

[10] Siddharth Choudhuri and Rabi N. Mahapatra. Energy characterization of filesystems for diskless embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 566–569, New York, NY, USA, 2004. ACM.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] Jonathan Corbet. zcache: a compressed page cache. `http://lwn.net/Articles/397574/`, 2010.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001.

[14] Digi-Key Corporation. Everspin technologies inc part number mr2a08ays35 mram 4mbit 35ns 44tsop. `http://www.digikey.com/` search keyword `mram`.

[15] Intel Corporation. Intel turbo memory with user pinning. `http://www.intel.com/design/flash/nand/turbomemory/index.htm`.

[16] Toni Cortes, Yolanda Becerra, and Raúl Cervera. Swap compression: Resurrecting old ideas. *Software—Practice and Experience (SPE)*, 30(5):567–587, 2000.

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

[18] Brian Dipert. Exotic memories, diverse approaches. *EDN*, April 2001.

[19] In Hwan Doh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Exploiting non-volatile ram to enhance flash file system performance. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 164–173, New York, NY, USA, 2007. ACM.

[20] Fred 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, January 1993. USENIX.

[21] Fred Douglis, Ramón Cáceres, Frans Kaashoek, Kai Li, Brian Marsh, and Joshua 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, November 1994.

[22] Nathan K. Edel, Ethan L. Miller, Karl S. Brandt, and Scott 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.

[23] Nathan K. Edel, Deepa Tuteja, Ethan L. Miller, and Scott A. Brandt. Mramfs: A compressing file system for non-volatile ram. In *MASCOTS '04: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 596–603, Washington, DC, USA, 2004. IEEE Computer Society.

[24] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, 21(2):194–203, 1975.

[25] Kylie M. Evans and Geoffrey H. Kuenning. A study of irregularities in file-size distributions. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 02*, San Diego, CA, july 2002.

[26] Y. Chen D. Niu Y. Xie Y. Chen G. Sun, Y. Joo and H. Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement, Jan 2010.

[27] Jean-Loup Gailly and Mark Adler. zlib 1.1.4. `http://www.gzip.org/`.

[28] N. Goyal and R. Mahapatra. Energy characterization of cramfs for embedded systems. In *International Workshop on Software Support for Portable Storage (IWSSPS) held in conjunction with the IEEE Real-Time and Embedded Systems and Applications Symposium (RTAS 2005)*, March 2005.

[29] Kevin Greenan and Ethan L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. pages 178–187, oct 2006.

[30] Mark Gritter. Your graph is bad and you should feel bad. `http://markgritter.livejournal.com/494915.html`, 2008.

[31] Nitin Gupta. compcache: Compressed caching for linux. `http://code.google.com/p/compcache/`, 2010.

[32] Dave Hitz, James Lau, and Michael 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, January 1994.

[33] Wen-Tzeng Huang, Chun-Ta Chen, Yen-Sheng Chen, and Chin-Hsing Chen. A compression layer for nand type flash memory systems. *Information Technology and Applications, International Conference on*, 1:599–604, 2005.

[34] Jung Soo Park Eunsam Kim Jongmoo Choi Donghee Lee Sam H. Noh Hwan Doh, Young Je Moon. In search of alternative uses of byte-addressable non-volatile ram:

A case study of a green web server cluster. In *Proceedings of the 1st Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH'09), co-located with ASPLOS'09*, Washington DC, USA, 2009.

[35] Bob Jenkins. A hash function for hash table lookup. *Dr. Dobbs Journal*, September 1997.

[36] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. Frash: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6(1):1–25, 2010.

[37] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, March 2003. USENIX.

[38] Kaminario. Kaminario homepage. `http://www.kaminario.com/`, 2010.

[39] Yangwook Kang and Ethan L. Miller. Adding aggressive error correction to a high-performance flash file system. oct 2009.

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

[41] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, New Orleans, LA, January 1995. USENIX.

[42] Jin Kyu Kim, Hyung Gyu Lee, Shinho Choi, and Kyoung Il Bahng. A pram and nand flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 31–40, New York, NY, USA, 2008. ACM.

[43] Michael Larabel. Using disk compression with btrfs to enhance performance. `http://www.phoronix.com/scan.php?page=article&item=btrfs_compress_2635&num=1`, 2010.

[44] Brian Leonard. Zfs compression - a win-win. http://blogs.sun.com/observatory/entry/zfs_compression_a_win_win, 2009.

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

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

[47] Robert Maloney. Gigabyte i-ram storage device. http://hothardware.com/Articles/Gigabyte-IRAM-Storage-Device1/, March 2006.

[48] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[49] T. Mikolajick, C. Dehm, W. Hartner, I. Kasko, M.J. Kastner, N. Nagel, M. Moert, and C. Mazure. Feram technology for high density applications. *Microelectronics Reliability*, 41(7):947–950, 2001.

[50] Ethan L. Miller, Scott A. Brandt, and Darrell 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.

[51] 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*, February 2004.

[52] Markus F.X.J. Oberhumer. LZO data compression library 1.0.8. http://www.oberhumer.com/opensource/lzo/.

[53] Kevin Parrish. Silverstone device gives 70% hdd boost. http://www.tomshardware.com/news/Storage-HDD-SDD-Cache-Drive,9591.html, February 2010.

[54] Hannes Payer, Marco A.A. Sanvido, Zvonimir Z. Bandic, and Christoph M. Kirsch. Combo Drive: Optimizing cost and performance in a heterogeneous storage device. In *Proceedings of the 1st Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH'09), co-located with ASPLOS'09*, Washington DC, USA, 2009.

[55] J. T. Robinson C. O. Schulz T. B. Smith M. Wazlowski R. B. Tremaine, P. A. Franaszek and P. M. Bland. Ibm memory expansion technology (mxt). *IBM Journal of Research and Development*, 45(2):271–285, March 2001.

[56] Erik Reidel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[57] David Roberts, Taeho Kgil, and Trevor Mudge. Using non-volatile memory to save energy in servers. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 743–748, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[58] Drew Roselli, Jay Lorch, and Tom Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, June 2000.

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

[60] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 405–420, San Diego, CA, January 1993.

[61] Jongmoo Choi Donghee Lee Sam H. Noh Seungjae Baek, Choulseung Hyun. Design and analysis of a space conscious nonvolatile-ram file system. In *TENCON 2006. 2006 IEEE Region 10 Conference*, november 2006.

[62] Julian Seward. bzip2 1.0.2. `http://sources.redhat.com/bzip2/`.

[63] Anand Lal Shimpi. Seagate's momentus xt reviewed, finally a good hybrid hdd. http://www.anandtech.com/show/3734/seagates-momentus-xt-review-finally-a-good-hybrid-hdd, May 2010.

[64] Peter 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.

[65] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[66] Jinsun Suk and Jaechun No. Performance analysis of nand flash-based ssd for designing a hybrid filesystem. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 539–544, Washington, DC, USA, 2009. IEEE Computer Society.

[67] Texas Memory Systems. Rackmount ram storage products. http://www.ramsan.com/products/line2, 2010.

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

[69] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems: then and now. *SIGOPS Oper. Syst. Rev.*, 40(1):100–104, 2006.

[70] Everspin Technologies. Everspin technologies takes mram to higher densities with 16 megabit product introduction. http://www.everspin.com/PDF/press/2010_april_19_16mb_mram.pdf, 2010.

[71] 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, September 1999.

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

[73] Theodore Y. Ts'o and Stephen 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.

[74] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, 2006.

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

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

[77] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.

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

[79] Michael Wu and Willy 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, October 1994.

[80] Erez Zadok, Johan M. Andersen, Ion Badulescu, and Jason 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.

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

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

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