Design and Implementation of a File System with on-the-fly Data Compression for GNU/Linux

Praveen B, Deepak Gupta and Rajat Moona
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur,
INDIA-208 016
email: {deepak,moona}@iitk.ac.in

Abstract

Data compression techniques have long been assisting in making effective use of disk, network and other similar resources. Most compression utilities require explicit user action for compressing and decompressing of file data. However, there are some systems in which compression and decompression of file data is done transparently by the operating system. A compressed file requires fewer sectors for storage on the disk. Hence, incorporating data compression techniques into a file system gives the advantage of larger effective disk space. At the same time, the additional time needed for compression and decompression of file data gets compensated to a large extent by the time gained because of fewer disk accesses. In this paper we describe the design and implementation of a file system for the Linux kernel, with the feature of *on-the-fly* data compression and decompression in a fashion that is transparent to the user. We also present some experimental results that show that the performance of our file system is comparable to that of Ext2fs, the native file system for Linux.

Keywords: GNU/Linux, File System, VFS, On-the-fly Data Compression, LZRW1

Introduction

Data compression techniques have long been assisting in making effective use of disk, network and similar resources. Compressing the data before storing it on the disk or transferring it over a network reduces the amount of bandwidth required for the data transfer. *Compress* and *pack* for Unix; *pkzip*[1], *lha*[2] and many others for MS-DOS; and *gzip*[3] developed by the GNU software foundation are a few widely used utilities for this purpose. These utilities require explicit user intervention for compressing

and decompressing of data. *Doublespace* for MS-DOS[4], *Stacker*[5] for MS-DOS and Macintosh are utilities in which compression and decompression of file data takes place on-the-fly, and is transparent to the user.

When a file is compressed before being written onto the disk, in most of the cases it occupies a lesser number of disk sectors than for the corresponding uncompressed file. This leads to efficient usage of the available disk space, i.e., the same amount of disk space can be used to store a larger amount of data. It also results in faster disk access since we need to access fewer number of sectors. Thus by choosing a compression algorithm that is fairly fast, the time spent in compressiion and decompression of data can be compensated for by the time gained in disk access. This leads to increased effective disk space with not much loss in terms of performance.

In file systems such as Ext2, a disk partition is divided into multiple block groups for efficient allocations of disk blocks for inodes and data blocks of a file. If the file data is stored in compressed form, it requires a lesser number of disk blocks which leads to more effective grouping of the inodes and the associated data blocks.

We have designed and implemented a file system with the feature of *on-the-fly* transparent compression and decompression of file data for the Linux kernel. Linux is a Unix-like kernel that was first developed for Intel 80x86 platforms and later ported to other platforms. It was first implemented as an extension to the Minix operating system[6]. Many modules of the kernel were later recoded and newer versions were released with advanced features. Our implementation was done by modifying the Ext2 file system code[7] distributed with the GNU/Linux distribution. GNU utilities such as *mkfs*, *fsck*, *tunefs* etc., for managing the file system have also been modified accordingly.

Design Objectives

The compressed file system has been designed with the aim of incorporating the feature of *on-the-fly* compression and decompression of file data while keeping the performance comparable to other file systems.

The design has been guided by the idea that it should be possible to integrate our

file system into the Linux kernel, with minor modifications to the generic file system code.

We aim at providing this feature in a fashion that is transparent to the users. They should remain unaware of the fact that the file is stored in a compressed form on the disk. The size and other attributes of the file as seen by them should be the same as that of the uncompressed file. We also aim at providing binary compatibility with the existing programs. All the existing programs should work without any modifications or re-compilation.

Another aim of our design is to allow multiple compression techniques to be used simultaneously in the file system. Thus the option of selecting a suitable compression algorithm for a specific file at the time of creating that file should be provided.

File System Design

In this section we present various design decisions that we have made to achieve our objectives. Then we discuss the issues related to the implementation of our file system.

The Virtual File System

The Linux kernel contains a Virtual File System (VFS) layer similar to the one that was originally introduced in SunOS for supporting NFS[8]. VFS is an indirection layer which handles the file oriented system calls. When a process issues a file oriented system call, the kernel calls a function in the VFS code. This function handles the structure independent manipulations and redirects the call to a function contained in the file system specific code, which is responsible for handling the structure dependent operations.

Our file system can be easily integrated with the VFS. All through our design we aimed at making as little changes to the generic code as possible. Whatever little changes *are* made to this code only enhance its functionality. This means that code for other existing file systems can work unchanged with the changed generic code.

Compress What and How?

The data in a filesystem is stored on the disk using logical blocks. Logical blocks are an operating system provided abstraction of physical disk blocks. Typically the logical block size is taken as a fixed multiple of the physical block size.

One can choose to do the data compression at the file level, or at the logical block level. Compressing at file level file would lead to a heavy degradation of performance while reading and writing small parts of the file, since the whole file needs to be compressed for each write and needs to be decompressed for each read. If the file data is compressed at the logical block level, some disk space may be wasted if the size of compressed data for a logical block is not an integral multiple of the size of a physical block. Further this would require that the size of a logical block should now be allowed to vary from the size of one physical block to some maximum integral multiple of physical block size. That is, a logical block should now be allowed to occupy 1, 2 or more (upto some maximum) physical blocks. Another advantage of using a logical block as the unit of compression is that if a disk block gets corrupted, only the data corresponding to that logical block is lost. If the file were to be compressed as a whole, the entire data of the file might be lost even if only one block was corrupted.

The performance advantage of compressing data at the logical block level, in our opinion, far outweighs the difficulties associated with implementing it. Therefore we have to chosen to compress data at the logical block level. The logical block size is always as an integral multiple of the physical block size. However, the number of physical blocks required to store a logical block depends upon how well that block of data gets compressed. For example, a logical block of 4K size can occupy one, two, three or four 1K physical blocks depending on the amount of compression achieved. Thus the kernel must support varying logical block sizes. Changes were required to generic file system code of the Linux kernel to achieve this. These are the only changes that have been made to the generic code.

The logical block size (which is the same as the physical block size in the case of an Ext2 file system) has to be specified at the time of creation of the file system. In

the case of a compressed file system, both the logical and physical block sizes have to be specified at the time of its creation.

The compressed data is padded to fit into an integral number of physical blocks. In order to retrieve the decompressed data, the actual size of the compressed data (in bytes) is stored in the first two bytes of the compressed data.

The gain due to compression is not very significant in the case of very small files. Therefore, in our implementation, we only compress files which have a size larger than a certain threshold. The threshold can be specified at the time of creating the file system. Further, since directories are usually small in size and more frequently accessed, we have chosen not to compress directories. Other special types of files are also not compressed.

Buffering of Decompressed and Compressed data

When *on-the-fly* compression is incorporated into the file system, physical blocks for a logical block can be allocated only after the compressed size of that logical block of data is known. This means that a logical block of data cannot be placed on the buffer cache queues till it is compressed. This fact has serious design implications that are discussed below.

In case of asynchronous file writes, the actual writing of data onto the disk takes place in background by the *bdflush* daemon, which periodically checks the buffer cache for entries which are marked dirty and flushes them onto the disk. Hence, to support asynchronous writes in our file system, we either need to store the compressed data in the buffer cache or allow the compression to take place at the time when the buffers are flushed. The first alternative implies that decompressed data can no longer be buffered, since the buffer cache can have only one entry for a logical block. This leads to paying a heavy penalty for file reads, since for every read, data from the buffer cache has to be decompressed before being given to the user program. The physical disk block allocation cannot be delayed till the buffers are being flushed because these buffers cannot be put on proper queues until their size and associated physical blocks are known. Thus they cannot be processed by the *bdflush* daemon unless we modify

its code which we wish to avoid.

One solution to this problem is to first estimate and allocate the number of physical blocks that would be required to store a logical block after it is compressed. When the actual compressed size is known, either extra physical blocks need to be allocated or the surplus physical blocks need to be freed. This may lead to heavy disk fragmentation which is undesirable.

To overcome this problem, we use the two level buffering mechanism, as implemented in Linux version 1.3.51 [9] onwards, to buffer the decompressed and the compressed data. Buffering decompressed data retains the advantage of buffer cache, while buffering the compressed data is required to support the asynchronous writes.

As shown in Figure 1, when a file is being written, data from the user buffer first gets copied into the virtual memory (VM) pages associated with that file. The appropriate VM pages are identified based upon the pagesize and the offset at which data is being written. All the VM pages corresponding to a file are arranged in a doubly linked list. These VM pages act as the first level buffers. Data from these first level buffers is compressed and stored in the corresponding buffer cache entry which gets written onto the disk, either synchronously, or asynchronously by the *bdflush* daemon. The buffer cache thus acts as the second level buffer.

Similarly when the file is being read, data from the disk is first read into the buffer cache. It is then decompressed into the appropriate set of VM pages associated with that file. Any further reads on that logical block of data can thus be directly satisfied using these VM pages, without the need for decompressing the data or accessing the disk.

Thus, while the VM pages maintain the advantage of speeding up the reads, the buffer cache aids in allowing writes to proceed asynchronously.

Support for Multiple Compression Techniques

As stated in the design objectives, our goal is to make the process of compression and decompression transparent to the user. At the same time, flexibility should be provided so that the user can choose the file system behavior with respect to individual

files, if required.

One goal of our file system is to support multiple compression techniques simultaneously in the file system. The choice of choosing a particular compression technique is available both to the system administrator and the user.

The system administrator can chose the compression technique to be used at the time of mounting the file system. Any file now created in this file system will be compressed by the compression algorithm thus chosen.

However, a user can override this default compression technique for the files his program creates. He can decide whether these files are to be stored in the compressed form or not, and can also choose the compression technique that is to be used to compress them. New flags, that can be specified with the *open* system call have been added for this purpose. This option facilitates choosing an appropriate compression technique based on the file type and also the requirements of the user. Thus the compression technique used is an attribute of the file rather than the attribute of the file system, and hence this information hence is stored in the inode structure for the file.

It may be noted that any compression algorithm as well as its implementation must be throughly tested before being incorporated in the kernel. Otherwise critial data may be lost due to bugs in the compression code.

Currently we have implemented LZRW1[10, 11], LZW12 and Huffman[12] compression techniques.

Disk Block Allocation

After a logical block of data has been compressed into a buffer, we allocate the required number of physical disk blocks, associate them with the buffer and store them in the block entry table of the inode. For efficiency we allocate all the physical blocks for a logical block in a contiguous fashion on the disk. It then suffices that we store the address of only the first of these physical blocks as the physical block number of the buffer. This along with the size of the compressed data determines the number of physical blocks that correspond to this logical block of data. Contiguous allocation

of physical blocks for a logical block allows the kernel to read a logical block with a single disk transfer. We have used a goal block allocation policy similar to that of Ext2fs for allocating the disk blocks.

The disk is partitioned into multiple block groups in a way that is done in Ext2fs. A block bitmap is used to represent the availability of disk blocks within a block group. A bit value of 0 indicates that the corresponding block is free and a value of 1 indicates that the block is allocated.

The goal block allocation policy tries to allocate contiguous blocks on the disk to store a file. When disk blocks for a logical block of a file are to be allocated, the block following the last allocated block for the previous logical block of the file is designated as the goal block.

The algorithm for allocating disk blocks first searches for the required number of free blocks starting from the goal block. If the required number of free blocks are found starting at the goal block, the search ends.

In case the required number of free blocks are not found at the goal block, we search in the near vicinity of the goal block to satisfy the request. This search is limited to the next 64 bits of this block group.

If the request is not yet satisfied, we then search for a free byte in this block bitmap. Starting from the first bit of this free byte, a search is made backwards to locate the first 0 bit in this sequence of 0 bits. Starting from this first free bit, we allocate the required number of blocks. Searching for a free byte has the advantage that block allocation request for the next logical block of the file can be satisfied at the goal block.

If a free byte is not found, the entire block bitmap is scanned for the required number of free blocks. If this too fails, each of the remaining block groups is tried until we get the required number of contiguous free blocks.

Representation of Logical Block Addresses

The other issue we need to deal with is the way the disk addresses of logical blocks are represented in the block entry table of the inode. In the Ext2 file system, a 32 bit

number is used to represent a physical disk block. Since the size of a logical block is always the same for Ext2, this information does not need to be stored. For our file system, however, each logical block corresponds to a variable number of consecutive physical blocks. We represent this information in the block table in the following way.

Of the 32 bits that are used to store the physical block number, we allocate n bits to store number of physical blocks corresponding to the logical block and the remaining $32 - n$ bits to store the first physical block number (Figure 2. The value of n depends on the logical and physical block sizes chosen. If we choose a logical block size of L bytes and the physical block size of P bytes, then the maximum number of physical blocks needed to store L bytes of data is L/P. Thus we need $n = \lceil \log_2 L/P \rceil$ bits to store the number of physical blocks required. This implies that the maximum total file system size is less than that of Ext2fs. But in practice however, since these numbers are enough to represent a very large file system, this does not make much of a difference. For example, if the logical block size is 8KB and the physical block size is 1KB, n is 3 and 29 bits are used for the physical block number. Thus disk partitions of sizes upto 2^{39} bytes can be handled.

Implementation of the File System Interface

File system related system calls in Linux have a file system independent portion and a file system specific portion of the code. In this sub-section, we discuss the file system specific issues of *mount*, *open*, *read* and *write* system calls.

Mounting a file system

The file system specific portion of the code for mounting a file system involves processing any file system specific options to the mount call and then reading the super block of the file system being mounted. The compression technique that will be used as a default for any newly created files in the compressed file system can be be specified at the time of mounting the file system.

A new <string=value> pair is added to the mount options for this purpose. In order to specify the compression technique to be used, the string "compress=<COMPR_ALGO>"

is to be passed as a mount time option. Based upon the value of <COMPR_ALGO>, the ID of the specified compression technique is stored. This value is then stored in the inode of any newly created files in the file system. Currently the value of <COMPR_ALGO> can be one of `lzrw1`, `huffman` and `lzw12` which correspond to the LZRW1, Huffman and LZW12 compression techniques respectively.

Opening a new file

The file system specific portion of the *open* system call just chooses a compression technique if the file is being created. New open flags allow the default compression technique specified with the *mount* call to be overridden, thus allowing multiple compression techniques to be used simultaneously in the file system. Using these flags, the compression technique to be used for the file being created can be specified. Following is an example of a call to the *open* system call for overriding the default compression technique.

```
fd = open("/tmp/example", O_CREAT|O_HUFFMAN_COMPR, 0644);
```

In the above example, `O_HUFFMAN_COMPR` could have been replaced with `O_LZRW1_COMPR` for LZRW1 compression or with `O_NOCOMPR` for no compression.

The default compression technique cannot be overridden for a file created using the *creat* call since the *creat* call does not have flags and adding an additional argument may cause the existing programs to stop working.

Reading from a file

The read system call in Linux gets directed to the *generic_file_read* routine, which is a file system independent interface for reading the data from the VM pages associated with the file into the user buffer[13]. The process of updating the VM pages with data from the buffer cache is implemented as a file system specific routine, which performs the following actions.

1. If the appropriate buffer cache entry is found and is up-to-date, data from the buffer cache entry is transferred into the appropriate VM pages For uncompressed files this implies a simple copy, while for compressed files this implies

decompressing the data from the buffer cache into VM pages. The exact size of the data that is to be decompressed (in bytes) is available in the first two bytes of the compressed data. Data from these updated VM pages is then copied into the user buffer.

2. If the corresponding entry is not found in the buffer cache, data is to be read from the disk. For this purpose, an entry is created in the buffer cache and data from the disk is transferred into this buffer. For compressed files, this data is then decompressed into the corresponding VM pages, and then copied into the user buffer. For decompressed files, a direct copy from buffer cache to VM pages and then to the user buffer is carried out.

As can be seen, file data needs to be decompressed only when it is not present in the VM pages. Once the VM pages contain the data in the decompressed form, any further reads of the same data do not involve either disk access or the overhead of decompression.

Writing to a compressed file

Writing to a compressed file differs from writing to an decompressed file in the way disk blocks and the corresponding buffer cache entries are allocated. For a file that is to be stored in compressed form, the allocation of disk blocks for a logical block of the file can be done only after the data in the logical block is compressed. This is achieved in the following manner.

After determining the logical block number to be written for the current file offset, the block entry table of the corresponding inode is used to locate the physical disk blocks for this logical block. If no physical blocks have yet been allocated for this logical block, a buffer (of the size of a logical block) is first created for it. This buffer does not yet appear in any of the buffer cache queues. Data from the user buffer is first copied into the corresponding VM pages. The data from these VM pages is then compressed. In the allocated buffer, we write the size of the compressed data (in bytes) in the first two bytes, followed by the compressed data itself.

The size of the buffer is changed to two plus the size of the compressed data, rounded off to the next higher integral multiple of physical block size.

The number of physical blocks required to store this logical block on the disk is now known. The block allocation routine is used to allocate the required number of physical blocks for this logical block. The buffer is then placed on the appropriate buffer cache queues.

On the other hand, if the physical blocks for this logical block have already been allocated, the buffer cache is checked to find the corresponding entry. If an entry is found, it is updated by the data from the corresponding VM pages. The updated buffer size is then checked against the available size for this logical block on the disk. If the number of physical blocks allocated for this logical block are such that the updated buffer does not fit into them, a reallocation of physical blocks for this logical block is done.

Avoiding Compression of Small Files

One problem that is encountered in a compressed file system is regarding the storage of small files. Storing smaller files in compressed form does not give any effective advantage, since both compressed and decompressed data require the same number of physical blocks in many cases. Moreover, we need to pay the penalty for compressing and decompressing the file for writes and reads. Hence we followed an approach that allows us to store the smaller files in uncompressed form and compress only those files that are larger than a certain threshold.

The threshold value can be chosen at the time of creating the file system. All files whose sizes are less than the threshold are stored in uncompressed form irrespective of the compression mode specified at the time of mounting or at the time of file creation. Once the file size crosses the threshold, and the `NO_COMPR` flag was not specified while creating the file, we start storing the file in compressed form. This technique requires the threshold size to be less than the logical block size of the file system since otherwise the change from uncompressed to the compressed mode would require additional reads and writes.

Performance Measurements

The performance of the compressed file system is affected by two factors, the time lost in compression and decompression of file data, and the time gained because of lesser data transferred to/from the disk. Thus, by using a good compression technique which is fairly fast, the time lost in compression and decompression can be made up to some extent by the time gained because of lesser disk transfer required. Hence we hypothesized that the performance of our compressed file system would not be much worse than that of Ext2fs.

In order to validate this hypothesis, we ran benchmark programs and compared the performance of our file system with that of Ext2fs. In this section, we describe our experiments and the results.

We used a rather low end computer with a Pentium 133 MHz processor and 32 MB of main memory to run the benchmarks. An IDE hard disk was used. The disk had the following characteristics.

Number of cylinders = 4385

Number of sectors per cylinder = 63

Number of heads = 16

Disk RPM = 5400

We conducted experiments to measure the time required for reading and writing one logical block of data. We compared the statistics obtained for our file system with that of an Ext2 file system with the same logical block size.

For this purpose we created a compressed file system with the logical block size equal to 4K bytes and physical block size of 1K bytes. We also created an Ext2 file system with the same logical block size. We chose a mix of file types (C source files, binary executables, text files etc.,) with an aim of achieving reasonable averages for the compression ratio. We used the LZRW1 compression technique for compression and decompression of file data.

Parameters Measured

We have measured the following parameters while reading and writing files in our file system. These parameters were also measured for the Ext2 file system. All the values are the time taken for reading/writing a single logical block (4K). For each of these parameters, we have measured both the CPU time and the real time.

- **Time for Synchronous Writes**

  In order to measure this time, we created files of different types (C sources, binary executables, text files etc.,) in the compressed file system and in Ext2fs. We restricted the size of these files to 48K in order to avoid the overhead of indirect blocks. Then we measured the times by overwriting these files in synchronous mode and taking the average of the obtained timings. Measuring the times while overwriting the files is to avoid the overhead of writing the block bitmap and the inode bitmap onto the disk.

- **Time for asynchronous writes**

  In order to measure the time, the same setup as used for the synchronous writes was used, except that the files are written in asynchronous mode.

- **Time for unbuffered reads**

  In order to measure this time, we created several files of size 4K in the compressed file system and in Ext2fs. We then rebooted the system in order to ensure that the data of these files is not in the buffer cache. Now, each file was read once and averages of the measured times was taken.

- **Time for buffered reads**

  In order to measure this time we first read a file to ensure that it was in the buffer cache. The same file was then read multiple number of times. This was repeated for several files and the averages of the measured times was taken.

- **Amount of disk space saved**

  In order to measure the amount of saving in disk space achieved by using our

	Synchronous Writes		Asynchronous Writes	
	CPU Time	Real Time	CPU Time	Real Time
Compressed fs	296 μsecs	6852 μsecs	280 μsecs	1273 μsecs
Ext2fs	131 μsecs	6692 μsecs	122 μsecs	169 μsecs
Ratio of Comprfs to Ext2fs	2.25	1.02	2.29	7.53

Table 1: Times for Writes

	Unbuffered Reads		Buffered Reads	
	CPU Time	Real Time	CPU Time	Real Time
Compressed fs	102 μsecs	2106 μsecs	10 μsecs	79 μsecs
Ext2fs	45 μsecs	2086 μsecs	10 μsecs	76 μsecs
Ratio of Comprfs to Ext2fs	2.26	1.01	1.00	1.03

Table 2: Times for Reads

compressed file system as compared to the Ext2 file system, we created two file systems. The first file system was of Ext2 type and had a block size of 1K. The second one was a compressed file system with a logical block size of 4K and physical block size of 1K. We copied the `/usr` directory tree to both the file systems and used the *df* utility to find the number of disk blocks used in the two cases.

It was observed that the Ext2 file system used 81722 1K blocks while the compressed file system used 61073 physical blocks (of size 1K). Thus there was a saving of 25.3% in the disk space. This was inspite of the fact that out of the 8129 files copied to both of the file systems, there were 2805 already compressed files which occupied 10100 blocks in both file systems. Additionally, there were 3045 files of size 1K or less which our file system did not compress.

Results and Analysis

Table 1 presents the average CPU times and real times required for writing one logical block of data to the compressed file system and to Ext2fs. In case of synchronous

Time for Compression	Time for decompression
132 μsecs	51 μsecs

Table 3: Times for Compression and decompression of 4KB data

writes, the CPU time required for writing one logical block of data into the compressed file system is more than twice the time that is required for a similar write into Ext2 file system. This is because of the extra time spent in compressing the data before writing it onto the disk. The ratio of the real times however indicate that the time lost in compression of file data is compensated to some extent by the time gained because of lesser disk transfer involved.

In case of asynchronous writes, the CPU times and the real times indicate that the compressed file system performs badly in comparison with Ext2fs. Since disk transfer in this case takes place in the background by the *bdflush* daemon, and the compression takes place in the foreground, the time lost in compression cannot be compensated for, to a significant amount, by the time gained by lesser disk transfer.

Table 2 presents the average CPU times and real times required for reading one logical block (4K) of data from the compressed file system and the Ext2fs. As seen, the CPU time required for an unbuffered read of 4K data from the compressed file system is over two times that required for a similar read from Ext2fs. This is clearly because of the extra time spent in decompressing the data. However, the ratio of real times for both the file systems show that the time lost in decompression is almost compensated for by the time gained because of lesser disk transfer involved.

In case of buffered reads, since the decompressed data is already available in the virtual memory pages, the CPU times and the real times for the compressed file system are equal to that of Ext2fs.

Table 3 presents average CPU times taken for compressing and decompressing one logical block (4K) of data using the LZRW1 compression technique. We chose a mix of file types (C source files, binary executables, text files etc.,), compressed the file data in chunks of 4K and decompressed it back, to obtain the average values for compression and decompression.

Table 3 also shows that most of the difference between the CPU times taken by Ext2 and our file system for reads and writes can be attributed to the CPU time required for decompression and compression respectively. For example, the difference between the CPU time for write in the case of Ext2 and the compressed file system is

about 160 μsec which is slightly more than the average CPU time required (132 μsec) for compressing a 4K block of data. Thus if more optimized, hand-coded assembly implementations of the LZRW1 compression and decompression algorithms are used, the difference between the performance of Ext2 and our file system can be reduced substantially. Techniques for optimizing the LZRW1 algorithm in assembly are well known (see, for example, Reference [11]).

From the results presented above, we see that in case of reads the performance of the compressed file system is almost as good as that of Ext2fs. While synchronous writes are also as fast as that of Ext2fs, the performance loss is significant in case of asynchronous writes.

Since reads typically constitute a large majority of the file system operations, and since most of the read requests are satisfied by the data readily available in the virtual memory pages, the overall performance of the compressed file system is not much worse as compared to that of Ext2 file system.

Conclusions

In this paper, we have presented the design and implementation of a Linux file system with on-the-fly compression and decompression of file data.

We started with the hypothesis that by using an efficient compression technique which is fairly fast, the extra time spent in compression and decompression is compensated to some extent by the time gained because of lesser disk access. Our experiments have validated this hypothesis and have shown that the performance of our file system is comparable to that of Ext2fs. Performance of our file system clearly depends, to a very large extent, on the ratio of CPU speed to disk speed. Since processor speeds are increasing much more rapidly than disk speeds. the gap between the performance of our file system and that of usual, uncompressed file systems will decrease further. The experiments have also shown that the choice of a logical block as the unit of on-the-fly compression is a reasonable one.

Hence, though disk space is not at premium these days, going for a compressed file system to get the advantage of increased effective disk space at a little cost in

terms of performance may not be a bad idea.

References

[1] PkWare. pkzip. http://www.pkware.com/

[2] SimTel Collection. lha. ftp://ftp.cdrom.com/pub/simtelnet/msdos/arcers

[3] GNU. gzip. http://www.gzip.org/

[4] HewlettPackard. Doublespace. http://hpcc920.external.hp.com/isgsupport/cms/docs/lpg12045.htm

[5] Macintosh. Doumentation on stacker. http://www.stac.com/pss/techmac.html

[6] A Tanenbaum, *Operating Systems: Design and Implementation*, Prentice Hall India, 1987.

[7] Remy Card, Theodore Ts'o, and Stephen Tweedie, 'Design and implementation of the second extended file system', *Proceedings of the First Dutch International Symposium on Linux* (1990).

[8] Sandberg.R, D.Goldberg, S.Kleiman, D.Walsh, and B.Lyon, 'Design and implementation of the sun network filesystem', *Proceedings of the USENIX Conference*, 119–131 (Summer 1985).

[9] Michael Elizabeth Chastain. Kernel change summaries for linux releases. ftp://ftp.shout.net/pub/users/mec/kcs/v2.0/

[10] Ross N. Williams, 'An extremely fast Ziv-Lampel data compression algorithm', *IEEE Computer Society Data Compression Conference*, 362–371 (1991).

[11] Ross N. Williams. Dr. Ross's Compression Crypt. http://www.ross.net/compression for up-to-date information on the LZRW* series of algorithms.

[12] Mark Nelson, *The Data Compression Book*, M&T Books, New York, 1992.

[13] Michael K Johnson. Linux kernel hackers' guide, version 0.7. ftp://sunsite.unc.edu/pub/Linux

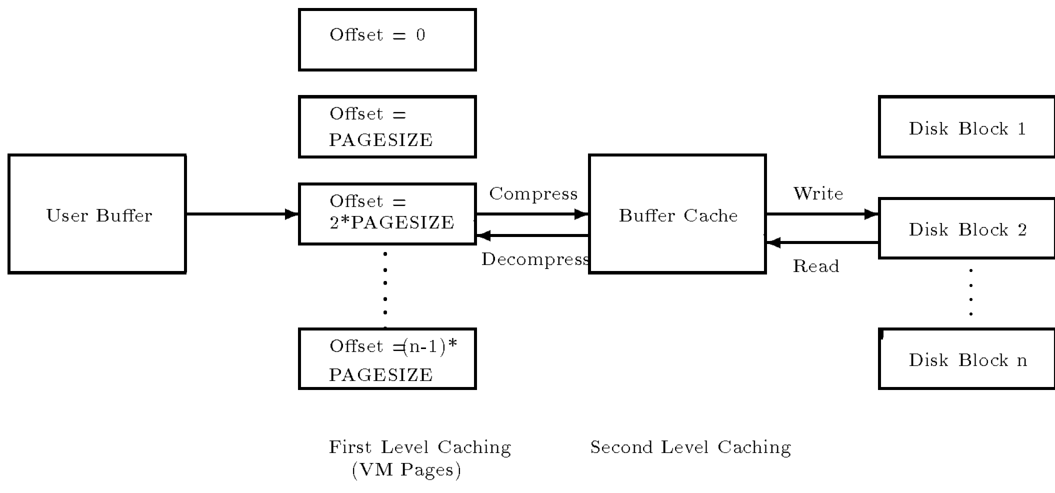

Figure 1: Buffering Mechanism in Linux

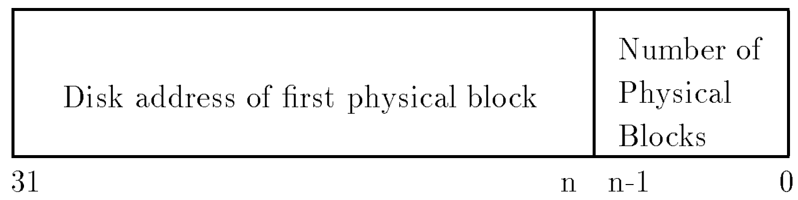

Figure 2: Representation of the disk address of a logical block in an inode.