The Nucleus **49**, No. 2 (2012) 129-137

The Nucleus
A Quarterly Scientific Journal of Pakistan
Atomic Energy Commission
NCLEAM, ISSN 0029-5698

TRANSPARENT COMPRESSION SCHEME FOR LINUX FILE SYSTEM

M. A. BASIR and *M. H. YOUSAF

Department of Computer Engineering, University of Engineering and Technology Taxila, Pakistan

(Received March 21, 2012 and accepted in revised form June 07, 2012)

Data footprint in data centers is growing day by day. Servers being used in research institutions, commercial web hosting companies and other enterprise server environment are now dealing with more and more bytes. Most of these servers have Linux operating systems and Ext2/3 comes as default file system in latest Linux kernel. A high-end compression system could be introduced in Ext2/3. It will be helpful in reducing data footprint in Linux based server environment. With slight modifications and using the sparse block feature of Ext2/3 file system transparent compression could be achieved. The reduced data footprint will result in reduced power consumption due to lesser disk reads and writes, and of course less data traffic on NAS(Network Attached Storage) and SAN(Storage Area Network). The reduced hardware might result in less management cost and of course less space for hardware.

Keywords: Linux File System, Ext2/3 Compression Support, Kernel Transparent Compression, Reduction of Storage Hardware Cost.

1. Introduction

Organization whether from IT or other sector are facing common challenge of managing an ever increasing amount of data and storage. With growth of softwares and hardware technology more and large files are being stored on servers for long period of time to meet compliance and other requirements, hence resulting in data footprint growth.

Microsoft's NTFS (File System), IBM's ZFS, Reser4 File System and JFS supports the transparent compression [1, 2]. The Windows NT Technology and Later operating Systems (like Server 2003, Server 2008, Vista and Windows 7) provide the user transparent compression and Encryption. Data Centers and other well known Linux kernel based operating systems like FreeBSD, Fedora and Ubuntu uses Linux File System Ext2/3 as default file system for formatting volumes [3, 4, 5]. Data Centers uses NAS devices, NAS device has the operating system which is diminished version of the whole operating system. FreeNAS or Open source NAS [6] is being used as operating system on NAS, DAS devices. Makatos et. al. proposed transparent compression to improve storage space efficiency and SSD-based I/O caches[7, 8].

There are some block compression techniques which are being used by ZFS and NTFS file systems. Microsoft's NTFS compression engine uses a modified form of LZ77 or LZ1 which is named as LZNT1. It uses a constant buffer size of 64K to compress the data transparently. This (small buffer size) causes more fragmentation in file system and makes it a bad choice to be used on server where storage IO is at high rate. On the other hand IBM's ZFS manages the problem of fragmentation with varying block size for compression. Unlike NTFS it uses different size of block for compressing data in different files and directories.

NTFS is not the common choice as file system in NAS servers. As stated NAS Server uses Free BSD operating System, it has EXT as the default file system to hold the data. So Linux LVM and its Software RAIDs are more common in Data Centers Systems and being used as default file system in Linux Operating Systems. So the major area of this research project focuses on EXT2/3 file system EXT 2/3 has no officially announced kernel support which provides the File System transparent compression. At the time of writing of this paper EXT has compression as a feature-under-test. 3rd party tools are available like e3Compr [9] which could be used as patch.

* Corresponding author : haroon.yousaf@uettaxila.edu.pk

Here is the Linux kernel storage subsystem, the area of interest for this research is *Logical File System* subsystem.

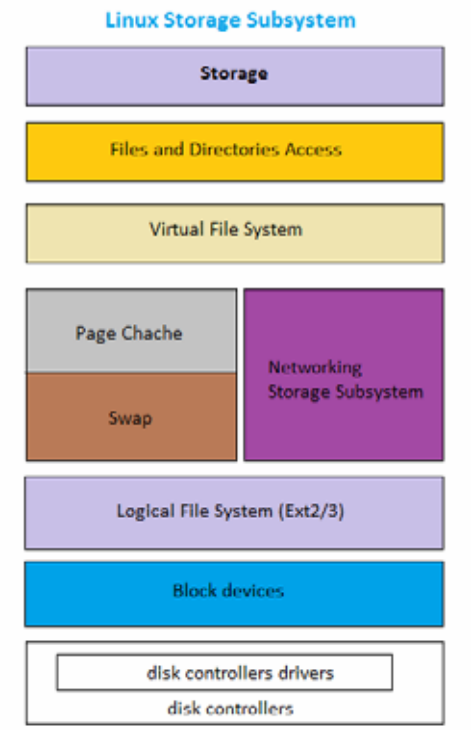

Figure 1. Linux Kernel Storage Subsystem.

Underlying sub-headings explains EXT system on some introductory level. Understanding of EXT data structure and its data addressing techniques are required to lay down architecture of new compression system

1.1 EXT Evolution

Ext2 file system was designed and release α in 1993 as Linux Kernel part. Authors of this file system are Remy Card and Stephen Tweedie. The design of file system is flexible enough to incorporate new changes. Ext3 the newer version of Ext2 was released with journaling support. Ext4 is the latest version of the file system with some further improvements.

Main architecture and data structures of Ext3are quite similar to Ext2 except that of journaling support (and of course some other features). The modification in Ext File System proposed here is equally applicable to Ext3 and Ext2 [3].

1.2. Structure of EXT

Linux EXT is rather simple file system, it tries to keep the disk reads minimum for a file (minimizes fragmentation). The basic read write unit on EXT is *block*. *Block* is grouping of disk sectors. The size of the sector depends upon the underlying storage hardware . In most of the storage devices it is of 512 bytes. So the block size in Linux is 1KiB, 2KiB, 4KiB and 8KiB. Most of the Linux Operating System implementations limits the block size to 4KiB.

Whole of the EXT partition is divided into group of blocks name *Block Group.* Block Groups are of same size except the last block. The block distribution on the disk is shown in Figure2.

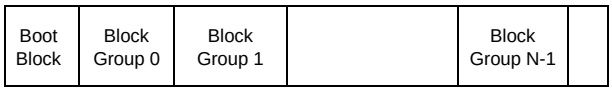

Figure 2. Disk layout of EXT partition

Boot Block stores different parameters of EXT file system, without boot block the file system could not be mounted by Kernel. File system keeps multiple copies of boot block in Block Groups. These copies of Boot Block are accessed only if the primary copy of the Boot Block found corrupted or damaged. Copies of Boot Blocks could be found in Block Group 0, 1, 3, 5, 7 and 9. Boot Block is also named as *Super Block* in Linux file system context [10].

Block Group consist of couple of data structures. Figure 3 shows the detail about these data structures.

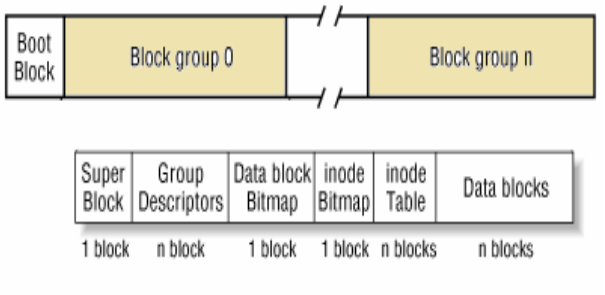

Figure 3. Block Group [10].

Super Block as described holds primary information about file system. The primary Super block is present at byte offset 1024 in EXT partition. The copy of Block group is present at byte offset zero of that block. The Block group which has no copy of super block fills zeros in that place.

Group Descriptors Holds the information about count of free *inodes*, count of free blocks, count of directories in current group, block address of *inode bitmap*, block address of *block bitmap* and block address of *Inode Table.* EXT keeps multiple copies of this data structure for reliability.

Data block Bitmap every single bit in this buffer specifies the usage of block, if bit is set to 0 then block is free, if bit is set to 1 then block is under use of file system. EXT tries to fit data block bitmap in one block. This rule could be used to calculate the size of Block group in bytes. The equation below can be used to calculate Block group size

BlockGroupSize = (BlockSize * 8) * BlockSize....(1)

For instance if BlockSize = 1K (1024 bytes) the BlockGroupSize would be *8388608 bytes* or *8MiB.* File system tries to keep the size equals to one block size.

Inode Bitmap is assigned integer array just like Data block bitmap, it can address the *BlockSize * 8* number of inodes. If bit is set then inode is under use.

Inode Table is an array of *Inode* data structure. The start of Inode table is pointed by Group Descriptor data structure. Every information related to the file or directory is stored in inodes. It stores size of files, different time stamps like create date, modified date, accessed date and deletion date. It holds the information about the chain or list of the file blocks. Primary focus of this research to perform detailed analysis of this portion of EXT file system. Detailed structure of *inode* could be seen in kernel documentation.

1.3. Data Addressing Mechanism

EXT uses four different modes of addressing blocks of the file. Pointers of the first 12 blocks of file are stored directly into the inode. Next three block pointers are used for *indirect*, *double indirect* and *triple indirect addressing.* These different mode of addressing are shown in the Figure 4. (taken from Linux Kernel documentation).

2. Background On File System Compression

Reiser4, NTFS, ZFS and JFS integrates compression mechanism in their architecture. Their compression mechanism is studied and analyzed to modify Ext2/3 for compression support. This section briefly highlights the mechanism of compression used by these file systems.

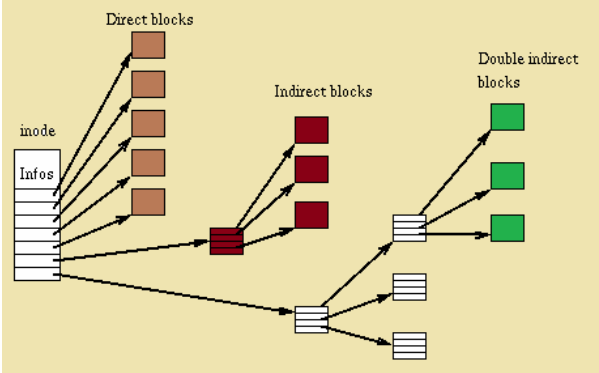

Figure 4. Block pointer addressing.

2.1. NTFS (New Technology File System)

2.1.1 Structure and Operations

To understand how the compression in NTFS works it is necessary to have a glance on the disk layout of NTFS. The file system uses Little endian format to save the information on disk. In NTFS everything is file. Unlike FAT even the file system meta data is placed in files. NTFS could be partition in following logical blocks.

Figure 5. NTFS disk structure layed down on disk.

The *bootblock* defines different parameters for file system. Like the sector size in form of bytes per sector. It defines *cluster size* of the file system which specifies the count of the sector in per cluster. Cluster size is logically equivalent to block size in Linux file systems. Cluster size could be calculated as

ClusterSize = BytesInSector * SectorsInCluster (2)

ClsuterSize is always calculated in bytes. Please refer to equation (2) above [11]. Boot block also specifies the start of the meta data files, NTFS maintains the backup of meta data files, the starting position of backup of meta data is also specified by the boot block. Boot block also contains the boot code which is being used by boot able partition to load the operating system.

Meta data in NTFS is saved in MFT records, MFT is the main record like inode in Linux file systems which hold every information about the file, like it can save hard links, file data block information to retrieve data about files, flags which shows different attribute associated with the file record. It can hold the data stamp information like date last modified, date last accessed and date created. It also hold the information about the file size.

Directories are saved in special blocks called Index blocks. NTFS uses B* trees for efficient sorting and retrieval of files for large directories.

Files block chain is maintained in data runs in MFT, Ext2/3 maintain this list in inode and uses direct, indirect, double indirect or triple indirect modes. Data runs is the special feature of NTFS file system to retrieve the file blocks information with minimal disk reads. This adds performance edge to NTFS.

2.1.2. Compression in NTFS

Main point of interest for NTFS analysis lies in understanding how it's compression works. Mentioned earlier that Microsoft has implemented a variant of LZ77 or LZ1 and named it LZNT1[11].

Cluster size here is important, NTFS uses cluster size of 4KiB for compression. NTFS groups further cluster to perform compression. The default and recommended size for compression is 16 clusters. So the total size of buffer which will be addressed during compression becomes. Please have look at equation (3) [12] below.

compression_block= 4KiB * cluster_group_size (3)

Where clsuter_*group*_size is 16 by default and recommended by Microsoft. *compression*_block here becomes 64KiB. The compression engine performs compression on this buffer and if new size after compression is less than 56KiB or 60KiB the compression on file is performed. Data runs in NTFS are updated after compression accordingly. How LZNT1 performs compression on buffer is out of the scope of this paper [3].

NTFS inserts the sparse run after the normal run (which contains the pointer for compressed data blocks). This pair of run for compressed file points to 16 Clusters. If the buffer of 64KiB is compressed to 48KiB, then first run will point to 12 compressed blocks of file and the sparse run will point to 4 virtual blocks of file, these 4 blocks would not be there on disk physically.

3. Advantages and Disadvantages

NTFS compression is good for files smaller than 40MiB, NTFS compression is not recommended for files which are accessed frequently. Due to 64KiB cluster the fragmentation in files increases significantly which reduces the performance of file system. NTFS compression does not work well on already compressed files like gz, zip, mp3 or other compressed files. NTFS is not recommended to be used on server class family operating systems like Windows 2003 and Windows 2008.

NTFS compression is good for PC based partitions where data is placed and backed up by user for archival purposes. It works good for small files and uncompressed files like wav, dat or text format files.

Due to cluster level or file block level compression the space inside one block is wasted and not claimed by file system. Like if a buffer of 4KiB is compressed to 3KiB the remaining 1KiB space in this black will be wasted. There is no mechanism in file system to reclaim this space.

3.1. ZFS (Zebra File Systems)

ZFS is the second file system selected to be analyzed and studied for its transparent compression technique. ZFS is one of the unique file system which integrates the concept of Logical Volume Manager and File System both.

3.2. Structure and Operations

According to the official document released by Sun Microsystems, Inc. ZFS consist of seven distinct subsystems. SPA (Storage Pool Allocator) , DSL (Dataset and Snapshot Layer), DMU (Data Management Unit), ZIL (ZFS Intent Log), ZAP (ZFS Attribute Processor) and ZVOL (ZFS Volume)[13]. SPA is responsible for Virtual device management in ZFS and is main point of interest for this research. Dataset and Snapshot Layer manages quotas, data integrity and reliability by using Checksum SHA-2. ZVOL manages the volume related work. In Linux this task is handled by Logical Volume Manager [13].

SPA manages virtual devices being used in ZFS and also take care of Block pointer and Indirect block pointer management. ZFS has two type of virtual devices (vdev), Logical and Physical Virtual Devices. Logical virtual devices are logical entities used to group up Physical virtual devices in device tree. ZFS attaches Vdev Labels with every virtual device and attach Labels to every device which is called Vdev Labels. ZFS ensures t-o replicate the Vdev labels which is 256KiB structure. This replication ensures that if one group of Vdev labels is overwritten then it can be recovered from other multiple copies[13].

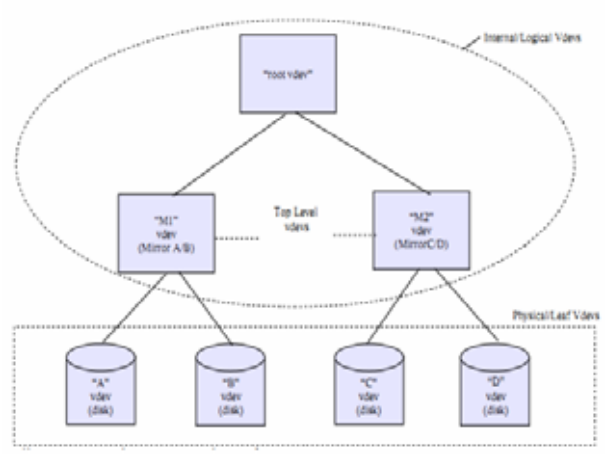

Figure 6. ZFS device tree [13].

After Vdev Labels from start of the file system there is 3.5 MiB reserved structure called Boot Block. Sun has reserved it for future user. There are two copies of Vdev Label at the end of the file system and two are placed at start.

Block Pointers in ZFS are responsible for handing over the file data to the requesting application. Block pointer is 128 byte structure. The table below specifies the detailed information about Block pointer structure.

4. Compression

Vdev1, Vdev2 and Vdev3 in above tables point to three different copies of data which are identical. Vdev is 32 it integer and offset is 63 bit integer value. It specifies the offset from start of the file system to the location where data lives. This is how ZFS implements its mirroring or data redundancy mechanism.

ZFS is unique in compression in a sense that it supports multiple algorithms for compression and specified by the *comp* flag in Block Pointer. It is of 8 bit, The table below shows value which could be used by this field.

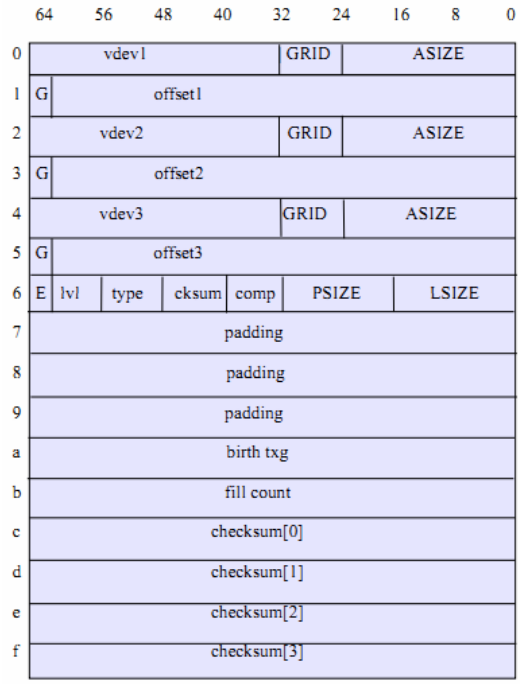

Figure 7. Block Pointer Structure [13].

Table 2. Compr byte values.

Algorithm	Value	Description
Lzjb	1	Compression On
None	2	Compression Off
Lzjb	3	Lzjb Algorithm

Size fields in Block Pointer structure has different significant meanings. LSIZE shows the size of the current block without compression, PSIZE shows the compressed size of the block. ASIZE is the size including all the metadata pay loads like gang headers or raid-Z parity.

LZJB is the lossless compression algorithm which is being used by ZFS. It is dictionary based algorithm and uses sliding window technique to analyze the whole buffer for compression. Discussion on compression algorithm is out of the scope of the research.

5. Advantages and Disadvantages

ZFS has over head of Volume management as this file system focuses on data integrity and uses

data mirroring or replication. This file system implements a good compression algorithm which is efficient in terms of CPU cycles consumption and compression. ZFS is not good for files under frequent use, like NTFS it is not good in compressing already compressed file formats like avi or mp3 formats.

ZFS is good for server class operating system and good in compressing file greater than 50 MiB to some files of TiB.

6. Proposed Architecture

After having some basic understanding of EXT, study and analysis of on-disk structure and compression for NTFS and ZFS this section will propose a new architecture for Ext2/3. The major objective and goal of this research is to perform least modification in the File system to achieve compression. This section will briefly describe the structure of Ext2/3 and then highlight the approach which will be used to add *transparent compression.* Some good design approaches will be adapted from the above described file systems. Figure 8 shows the sub system red where kernel could be modified.

6.1. Proposed Compression System

Inode structure has flag fields which define different attributes associated to the file. Some of the flag values are experimental and currently not supported by Linux kernel. These flags are compression flag, deletion and un-deletion. Compression flag here is used to specify that file is compressed or not. When compression on the fly is performed this flag in inode of every directory and file is set. When blocks for such file are accessed the special decompression mechanism is invoked which decompresses the file data and passes it to requesting application, this keeps the compression and decompression mechanism *transparent* to requesting applications.

The proposed architecture introduces the concept of *compression window.* Multiple blocks of file are collected to form a compression window. Compression window size is selected to be of 128KiB. For example, the file of 1MiB could have 8 compression windows of 128KiB. Compression window size is selected to be constant for all files in Ext2/3 partition.

Unlike ZFS which selects different size for compression window for different files or directories. NTFS selects the constant size for compression window for every file in the partition. Figure 9 highlights the concept of compression window introduced in this research.

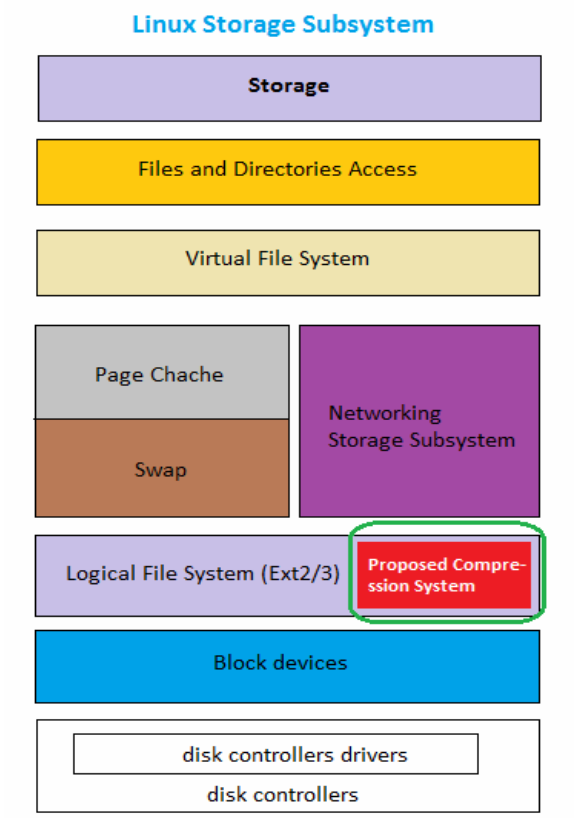

Figure 8. The Big Picture

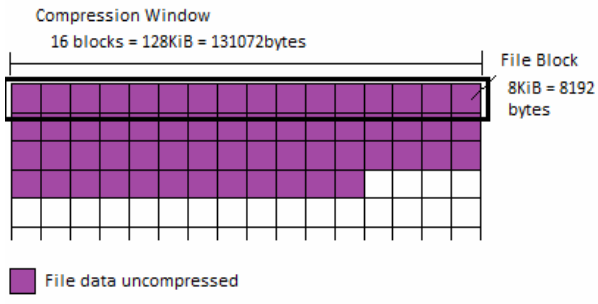

Figure 9. Compression window .

6.2. Working of Compression System

Main challenge is to keep the unique identification for every block of file even in compression window. A unique approach is devised to address the issue. Figure10 details the

example of compression with block size of 8KiB and file size of 479KiB, the compression window size selected for this test case is 128KiB.

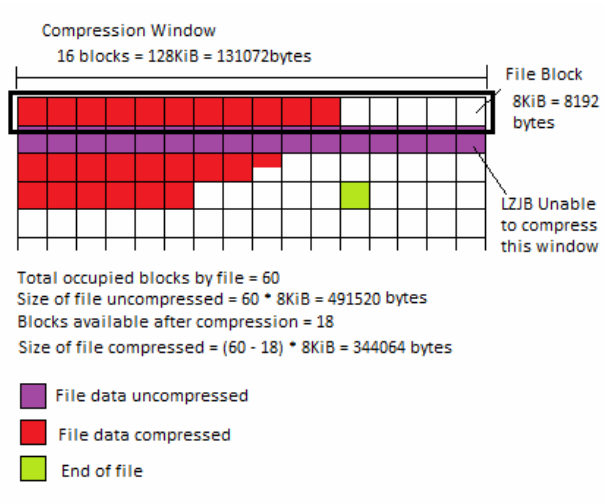

Figure 10. Compression example.

A file block *N* is compressed and its size after compression is calculated. Two bytes are reserved in front of every compressed file block which stores the compressed size of the current block. Next block is compressed, its size is calculated, padded in front of compressed buffer and then inserted after the *N-1th* block. This mechanism goes on till all of the blocks in compression buffer are compressed. *Two bytes allows to keep the track of blocks with maximum size of 65536 bytes which is 64KiB*. The maximum allowed block size in Ext2/3 is *8KiB*. Figure 11 elaborates the phenomena of accessing the compressed block in compression window.

After performing compression some blocks at the end of compression window are now available. The relevant inode pointing to this block is set *0x00000000.* Number of free inodes count in group descriptor of current Block group is update. This will let Kernel to use inodes and blocks which are available after compression. This step in Ext2/3 compression is necessary so that new block which are available after compression could be consumed by new files.

Sometimes lossless compression algorithm LZJB is unable to compress the whole compression buffer and its size grows even larger from uncompressed buffer. If the size of whole compression buffer grows from the compression window size then contents of that particular compression window are not updated on the disk and no inodes block reservation are updated.

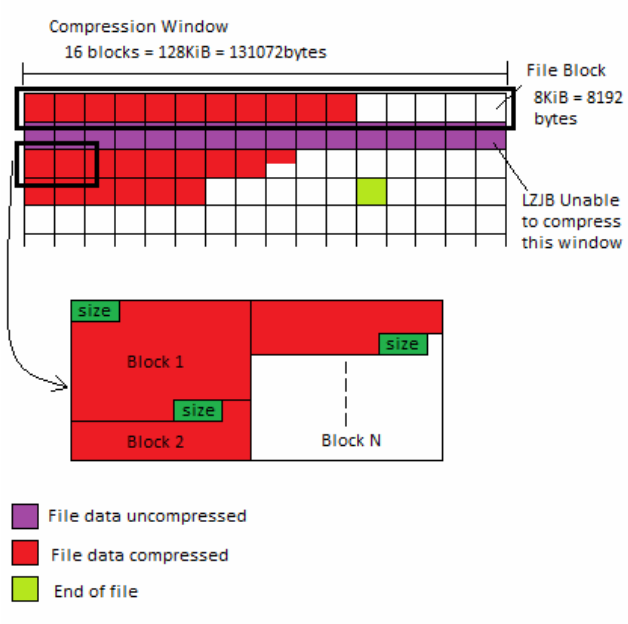

Figure 11. Accessing the unique file block after compression has been done on the file data.

For files smaller than 128KiB are assumed to have compression window equal to the size of file.

7. Experimentation Results

Compression for lossless algorithm is totally dependent on the nature of data. Compressed file formats like avi, gz or zip do not give good compression results.

Units being used for measuring compression system are *bit by byte ration* (bit by byte), *compression percentage* and *Kilobytes per second* [14]. First two units are used for measuring the compression power of the system. The later one is used for performance measuring of compression system. Optimizing the performance of this system is the future work for this research. The main focus is to design and propose a basic compression system for Ext2/3.

Results of proposed system are controlled by following factors.

- Compression window size
- File block size
- Nature of data (compressed or uncompressed)

- Fragmentation of File

Effect of compression window size and file block size were analyzed. Nature of data selected for this test run was uncompressed. A fresh EXT partition was created and data files were placed on the partition to execute the compression test. Following chart elaborates some detailed information about the findings of test run.

Figure 12 shows the effect of block sizes on compression. Bit by byte ratio is count of bit which could be used to represent on byte [14]. Bit by Byte ratio for a file of size 800bytes compressed to 100 bytes will be 100/800 * 8 = 1bpb.

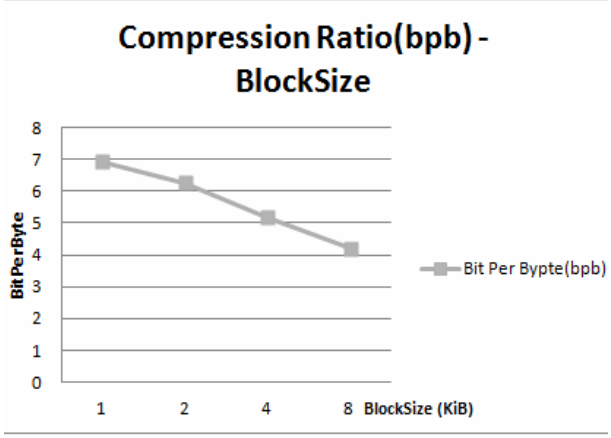

Figure 12. Block size affecting compression ratio measure in bits per byte.

Figure13 shows the effect of block size on compression in another representation name as compression percentage.

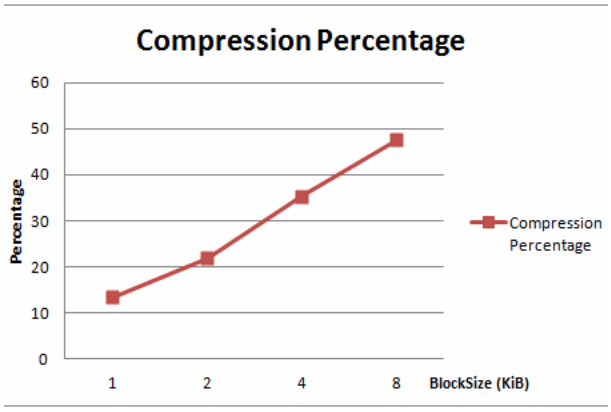

Figure13. Compression percentage plotted against block size.

Another interesting analysis of the compression system is to study bpb ratio for different window sizes. The detailed test run was executed to grab this information shown in Figure14.

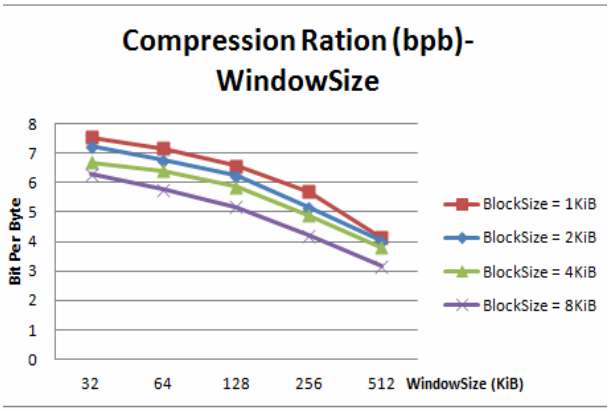

Figure 14. Compression ratio for different block sizes plotted against different window size selected for compression.

Compression percentage for different window sizes and different block sizes was analyzed. Experimentation results for this configuration are shown in Figure15.

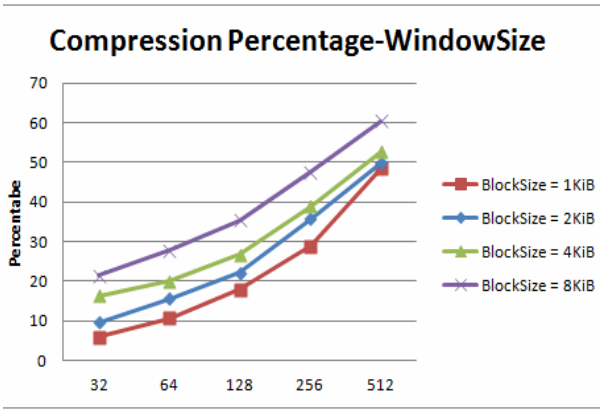

Figure 15. Compression percentage for different block sizes plotted against different window sizes.

8. Conclusion

The main objective of this research is to propose some new changes in Linux kernel and Ext2/3 to compression support, while keeping changes as minimal as possible. Well defined commercial and research projects exist for this purpose. But Ext2/3 has its own advantages and usability.

According to experimental results found, compression window size of 128 KiB with block size of 4KiB is highly recommended. Compression window with 512 produces high compression but on the cost of CPU usage. Ideally 50 CUP cycles are consumed for every byte to be compressed. As for NTFS and ZFS it is recommended not to compress those files which are in frequent use.

ZFS uses dynamic compression window for every file. Calculating that window size is also an over head on CPU. The propose architecture suggest to keep it to constant to 128 KiB. ZFS has other heads of volume management, quota management and data structures overhead. Ext2/3 has simpler data structures as compared to ZFS and of course causes less impact in terms of CPU usage.

NTFS implements LZ77 which is less efficient as compared to LZJB. Comparison of compression algorithm and their optimization is out of the scope of this research.

Performance optimization of proposed architecture is one of the future task. As this project mainly focuses on investigating existing systems and proposing some new structure for Ext2/3 to achieve compression. Adding compression support to encrypted file is also one of the task required to be implemented in future.

This research adapts one of the compression algorithm from ZFS. Designing the new compression algorithm for Ext2/3 is also required.

References

[1] NTFS On Disk Structure and Compression, Rel-Soft Technology (Oct., 2004).

[2] File System Comparison, WikiPedia, http://en.wikipedia.org/wiki/Comparison_of_file_systems (Feb. 27, 2012).

[3] TOMOYO, “Linux Kernel Implementation by TOMOYO which focuses mainly on security ”http://tomoyo.sourceforge.jp/

[4] JNODE, “Java New Operating System Design Effort”, http://www.jnode.org/ (May 14, 2012).

[5] Sources For Linux Kernel, “Linux Kernel Source Code”, http://www.kernel.org/pub/linux/ (May 3, 2012).

[6] Free NAS implementation, “Free NAS”, online at http://freenas.org/FreeNAS/ (May 15, 2011).

[7] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, “ZBD: Using Transparent Compression at the Block Level to Increase Storage Space Efficiency” International Workshop on Storage Network Architecture and Parallel I/O(SNAPI) (2010).

[8] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, “Using Transparent Compression to Improve SSD-based I/O Caches.” to appear in the ACM/SIGOPS European Conference on Computer Systems (EuroSys 2010).

[9] e2Compr, “EXT Third party compression system”, http://e2compr.sourceforge.net/ (April 3, 2011).

[10] Gregorio Narváez SANS Institute InfoSec Reading Room, “EXT3 Journaling File System Forensic Analysis” (December 30, 2007).

[11] Red Hat Linux 9: Red Hat Linux System Administration Primer Copyright © 2003 by Red Hat, Inc. (Feb 20, 2008).

[12] By Greg Schulz, “Application Agnostic Real-time Data Compression How Real-time compression across different applications reduces data footprint and energy efficiency without performance compromises,” (Feb. 11, 2008).

[13] 4150 Network Circle, Sun Microsystems, Inc. “ZFS On-Disk Specification” (Jan. 1, 2006)

[14] Arturo San Emeterio Campos, The introduction to Data Compression, http://www.arturocampos.com/cp_ch1.html#Measuring%20the%20compression%20rate, (July 6, 2000).