Compression and Replication of Device Files using Deduplication Technique

Bharati Ainapure
Assistant Professor
Department of Computer Engineering.
MITCOE, Pune University, India.

Siddhant Agarwal
Abhishek Somani
Department of Computer Engineering.
MITCOE, Pune University, India.

Rukmi Patel
Ankita Shingvi
Department of Computer Engineering.
MITCOE, Pune University, India.

ABSTRACT
One of the biggest challenges to the data storage community is how to effectively store data without taking the exact same data and storing again and again in different locations on the back end servers. In this paper, we outline the performance achievements that can be achieved by exploiting block level data deduplication in File Systems. We have achieved replication of data by storing a deduplicated copy of the original data on the backup partition, which can enable us to access or restore data in case of a system crash.

General Terms
Block Level Deduplication, Secure Hash Algorithm.

Keywords
Deduplication, Replication, Data Storage, VFS (Virtual File System).

1. INTRODUCTION
The kernel comprises of the following components:

1.1 Virtual File System
The Virtual File System (VFS) acts as an interface between the Kernel and the different file systems [5]. It is a Kernel software layer that handles all the system calls related to a standard UNIX file system. Linux manages multiple file systems through the use of VFS. It bridges the gap between different file systems. It provides a common system call interface to the userspace.

The VFS stores information related to a file system using several data structures:

1. Superblock: Stores information related to a File System.
2. Inode: Keeps track of files and stores metadata of the file.
3. File: Represents an open file for a particular process.
4. Dentry: Stores the recently used file system paths and inodes.

Thus, the purpose of the VFS is to allow the client applications to access different types of file Systems uniformly [5]. It acts as an interface between the kernel and the file systems.

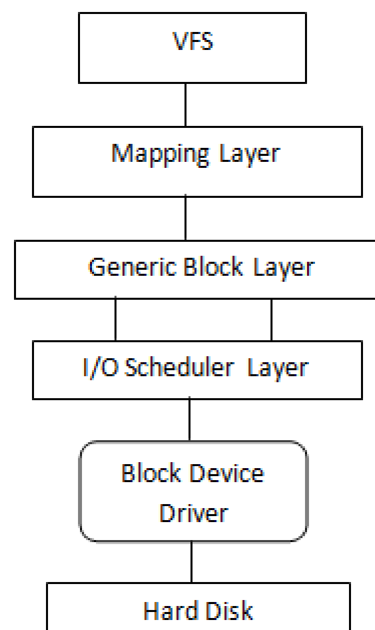

Fig1. General Kernel components

1.2 Generic Block Layer
The Generic Block Layer is the Kernel component layer that handles all requests for all the block devices in the system [6]. It helps to put the data buffers in high memory. Disk data is directly put in the user mode address space without being copied to kernel memory first using this address layer [4]. It manages the logical volumes.

1.3 I/O scheduler
The block device drivers is used to transfer a single sector at a time. But if the block I/O layer perform an individual I/O operation for each sector, it would lead to very poor disk performance [7]. So, the block I/O layer clusters several sectors and perform an operation on this cluster of sectors. It is done by the I/O scheduler. Whenever I/O operation is requested by the kernel, the request describes the requested sectors and the kind of operation to be performed, i.e. read or write, on them [7]. This request is not performed immediately, but is actually scheduled at a later time. Whenever new block data transfer is requested, the kernel checks the previous requests and if the head moment of the disk is less between previously stored request and the current one, then performs

the operation on the previous and the current one. Thus, performance is improved using the I/O scheduler.

1.4 Block Device Drivers:

In an I/O operation, the disk controller moves the heads on the disk surface to reach the correct position which requires significant amount of time[3]. But if the heads are correctly placed, the time required decreases and speed of the operation is increased. This is the key aspect of Block devices. The block device drivers get the requests from the I/O scheduler, and do whatever is required to process them. They may handle several block devices.

The Device Driver is an interface between high level computer programs and the Hardware device and communication between the two takes place with the help of computer bus [5]. Whenever a calling program invokes a routine in a driver, the driver issues a command to the device. Drivers are hardware independent and operating system specific. Drivers can be build either as part of the kernel or as loadable modules. The advantage of loadable modules is that, they can be only loaded when necessary thus saving the kernel memory [5].

Device files contain 3 main parameters:

1. **type:** block device or character device
2. **major number:** identifies the device type
3. **minor number:** identifies a specific device among a group of devices that share the same major number.

The mknod( ) is used to create device files. It receives the name of the device file, its type; it's major and its minor number.

To support a disk, we first need to register the device driver with the kernel [11]. For each disk, the driver allocates a gendisk and a request queue for each gendisk [2]. The gendisk structure is allocated using the alloc_disk(). The alloc_disk() allocates the array required for partitions. The partition of the disk is stored as an array of hd_struct. hd_struct represents the partition information. The driver needs to fill the partition information.

A device is read/written as a normal file. The kernel accesses these devices when the file system on the device is mounted.

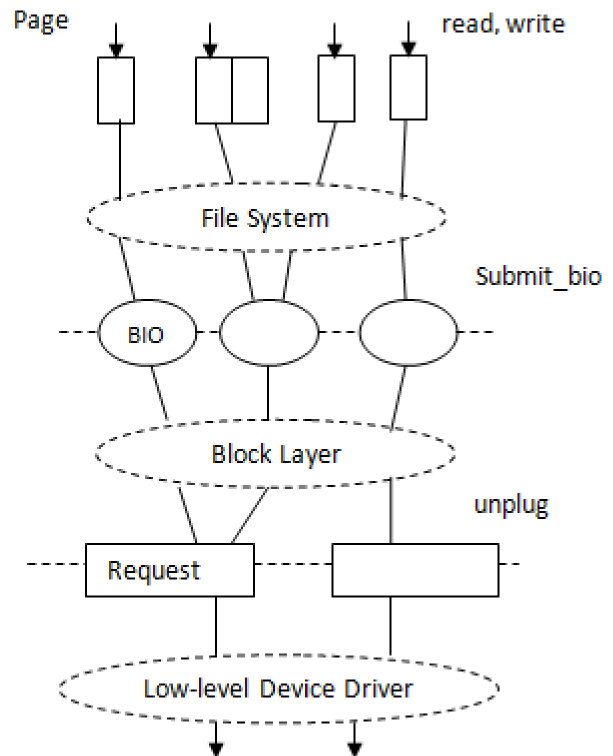

Fig2. Relationship between page, BIO and request

In the next section, we will go through the related work of the paper.

2. RELATED WORK

The references [1] and [7] brief us about the multipath request processed through the bio and the device mapper in the kernel. The read and write request is fist sent to the bio and the to the I/O scheduler for further processing.[2] and [6] inform us about the secure hash algorithm used to calculate checksum and deduplication of data. [3] helps us go through the different events geared towards those who are doing desktop application and infrastructure development,[4] explains the generic block layer. It also helps to go through the documentation file in the kernel. [5] briefs us about the communication between the user space and kernel space.[9] shows the linux kernel archives and helps to access the source code for linux kernel. [10] helps us find out the technologies for quantum that increases disk performance while reducing capacity needs. Reference [11] briefs us about the device files and its information. [12] and [13] help us understand the use of commands like dm_setup and dm_crypt to create a target device. The following references inform us about the various linux commands used for loading and removing modules in the kernel.

3. PROBLEM STATEMENT

In this paper, we represent our idea of how to get deduplication of Device Files. Here we show the processing of a read or write request for a block of data from or to the device files. The request is further processed through the various layers of the kernel[4]. To implement deduplication we then calculate the checksum on every block of data using the SHA1 algorithm. The checksum calculated is then stored in a mapping table in order to avoid storage of redundant blocks of data.

This stores only a single copy of the redundant block hence resulting in saving the storage space. We have implemented replication of the deduplicated data by storing it on the backup partitions. On receiving a block of data from the Generic Block Layer, a checksum is calculated. Each block device has a set of functions stored in its f_ops structure[3]. Our functions are mapped to those pointed by the f_ops, defined explicitly.. We can get this by using:

```
.read = device_read,

.write = device_write
```

Thus, the read () function of the device file is mapped by our read function, i.e., device_read() and the write() function by our device_write(). Deduplication is then done on the backup partition, while the original partition stores the entire data.

4. IMPLEMENTATION

We now see the implementation of our idea. The implementation is divided into following steps.

Step1:

We first register the block device driver by following function:

register_blkdev(major_num, "block_device");

Using **insmod,** we load the device driver in the kernel[11]. Now our device driver is ready to be used.

We use several data structures so as to efficiently perform the required operation. We discuss here some of the major data structures.

1. **buffer_head**: The buffers are tracked in the kernel using the buffer_head data structure[4].

Its structure is as shown below:

```
Struct buffer_head

        {
        Char b_data;     /*pointer to the
                                  buffer*/
        Unsigned long b_blocknr; /*logical
                block number */
        Unsigned long b_size; /*block
                                  size*/
        …
        };
```

Step2:

Whenever a file is requested for an I/O operation, it may contain the requested stream of data spread over several pages or several file system blocks[9]. If the data is stored contiguously on the disk, then the entire request can be represented by a single data structure, which can be carried out by the BIO data structure. The pages containing contiguous data are grouped into a single BIO[3]. The BIO data structure is used by all layers below VFS to represent the I/O for a file. Each segment in a BIO is represented by a bio_vec structure. Its fields are as shown below:

```
                Struct bio_vec

        {

        Struct page* bv_page;          /*pointer to the
                        page descriptor of segment's
                                  page frame*/

        Unsigned int bv_len;           /*length of
                        segment in bytes*/

        Unsigned int bv_offset;        /*offset of
                        segment's data in page frame*/

        …

        };
```

The Generic Block Layer starts a new I/O operation by allocating a new BIO descriptor through the bio_alloc () function. The Kernel keeps track of the bio_vec structures so that it is able to allocate the segment descriptors to be allocated in the BIO structure.

Step3:

The list of pending requests is maintained by each block device in its own request queue. Each physical block device contains one request queue. To increase the disk performance, I/O scheduling is performed separately on each request queue. The **request_queue** data structure contains the following fields:

```
Struct request_queue

        {

        Struct request list rq;        /*data structure
                used for allocation of request
                descriptors*/

        Unsigned int bv_offset;        /*offset of
                segment's data in page frame*/

        …

        };
```

The function blk_init_queue() is used to allocate the request_queue.

The I/O requests are represented by a BIO structure [7]. A BIO is received for each I/O request.

The BIO manipulations are performed in following ways:

1. Cloning: In this, a copy of the BIO is made[8]. The clone will have a completion handler, while the original won't have it[1].
2. Mapping: The cloned BIO is passed to the target driver. Here, the bi_bdev field is set to the device it wants to send the BIO to.
3. Completion: The BIO can be remapped or completed when the completion handler is invoked. When all the cloned BIO's are completed, the original BIO is said to be completed.

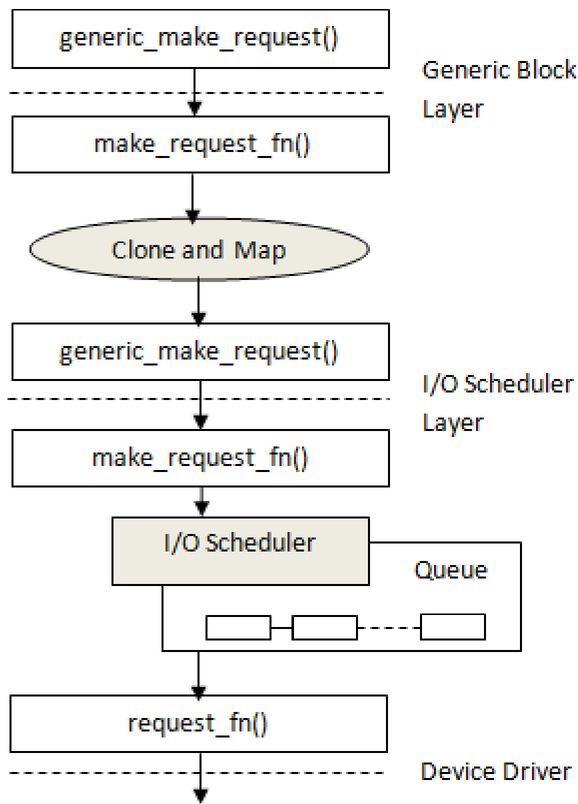

Fig. 3 BIO flow

Step4:

We have included the following data structures so as to store the checksum of the written data and to retrieve it from the backup partition.

1. *typedef struct hash_lba {*

   Unsigned long long hash_key;

   Unsigned long long sect;

   }hash_lba_t;

This structure stores the sector number in the 'sect' ield and the checksum in the 'hash_key' field.

2. *typedef struct hash_pba {*

   int address;

   unsigned long long hash_key;

   }hash_pba_t;

This structure stores the checksum in the 'hash_key' field and the offset of the sector in the 'address' field.

3. *typedef struct count_pba {*

   int cnt;

   }count_pba_t;

This structure stores the count of the number of times a block with the same checksum occurs in the file.

Using these structures, we can read the data from the backup partition.

Step5:

Now, whenever data is passed block by block into the device file, checksum will be calculated via our target and deduplication takes place.

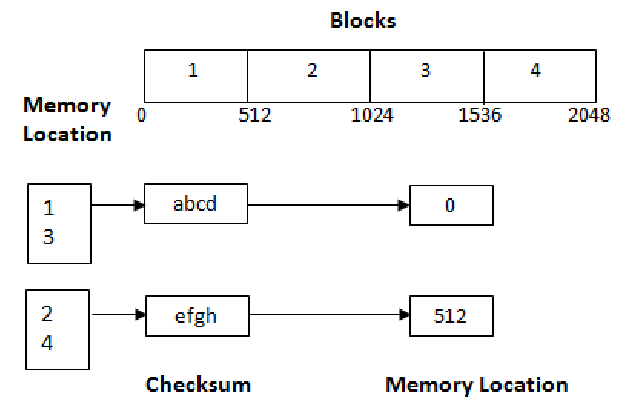

Fig 4. Checksum calculation

In the above figure, there are 4 blocks in memory of which blocks 1 and 3 are similar and 2 and 4 are similar. So, the checksum for the similar blocks is same[2]. So, all the blocks with similar checksum are pointed to a single memory location. Thus, the blocks 1 and 3 all point to memory location 0, which is the actual memory location of block 1. Similarly, the blocks 2 and 4, all point to the memory location 512, which is the memory location of block 2. Thus, out of 4 memory locations, only 2 memory locations are used, thus saving the memory by almost 50%.

Step6:

In this step, a copy of the deduplicated data is created and saved on another partition of the disk. This is done in order to access or recover data in case of a system crash. In replication, if the data changes in the original copy of the data, modifications should also be made in the duplicate one. This should be done properly so that the duplicate data is the exact replica of original data at any instant of time.

5. RESULT AND ANALYSIS:

The following graphs show the comparison in system performance with deduplication and without deduplication.

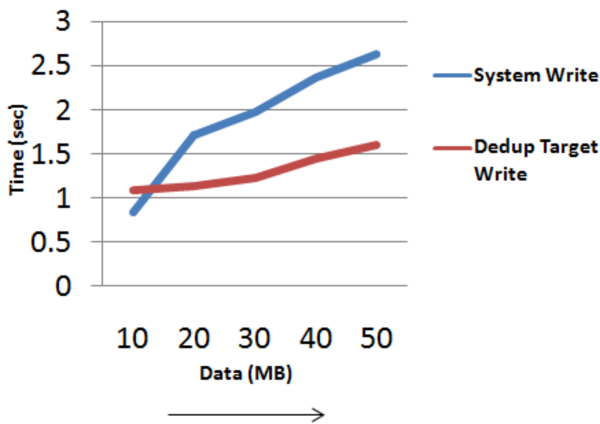

Fig 5. Graph for Write Request (Full Deduplicate Data)

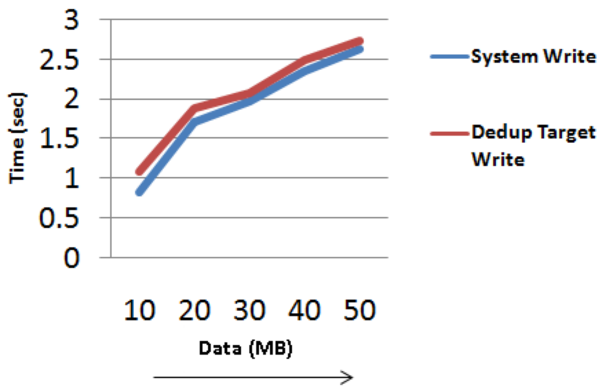

Fig.6 Graph for Write Request (Full Non-Deduplicate Data)

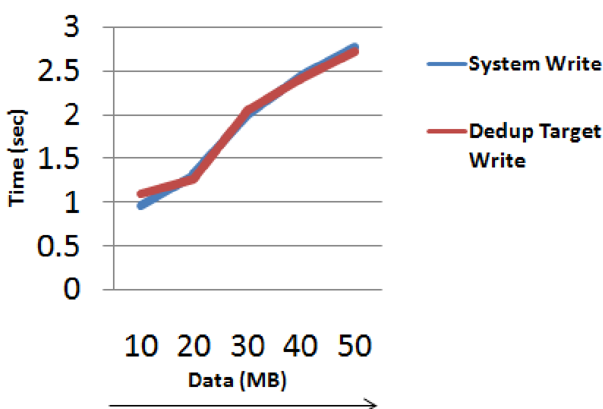

Fig.7 Graph for Write Request (Mixed Data)

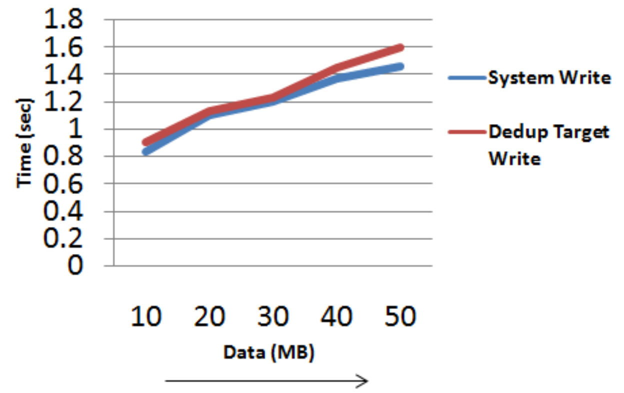

Fig 8. Graph for Read Request

6. ADVANTAGES:

1. Data Deduplication can achieve the compression ratios ranging from 10:1 to 50:1.
2. Deduplication is a good choice for data that is uncompressed and unencrypted.
3. Compression is useful for extending the life of older storage systems.
4. Replication can be used in case our system crashes. So, the crashed data can be recovered.

7. CONCLUSION AND FUTURE SCOPE

Removing the redundant data by removing the duplicate blocks of data and storing only the unique blocks leads to improve space efficiency to a great extent. In some cases, the space efficiency increases to about 80%, then that without deduplication. Also, if the system crashes, then it is possible to recover the data from the copy of it we created using replication. Thus, important data can be kept safe via replication.

In replication, if the data changes in the original copy of the data, modifications should also be made in the duplicate one. This should be done properly so that the duplicate data is the exact replica of original data at any instant of time. The Challenge is to achieve maximum dedupe ratio with as little effect on throughput as possible. Propose effective methods to estimate the opportunities of data reduction for large-scale storage systems. Reduce the data center footprint and thus reduce the power needs.

8. REFERENCES

[1] Request-based Device-mapper multipath and Dynamic load balancing by kiyoshi Ueda, Jun'ichi Nomura and Mike Christie.

[2] A Device Mapper based Encryption Layer for TransCrypt by Sainath Vella.

[3] Edward Goggin, Linux Multipathing Proceedings of the Linux Symposium, 2005.

[4] Jens Axboe, Notes on the Generic Block Layer Rewrite in Linux 2.5, Documentation/block/ biodoc.txt, 2007.

[5] Netlink - Communication between kernel and userspace (PF NETLINK). Manual Page Netlink.

[6] Satyam Sharma, Rajat Moona, and Dheeraj Sanghi. TransCrypt: A Secure and Transparent Encrypting File System for Enterprises.

[7] Mike Anderson, SCSI Mid-Level Multipath, Proceedings of the Linux Symposium, 2003.

[8] Daniel Pierre Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly& Associates, Inc., third edition, 2006.

[9] The Linux Kernel Homepage, Website http://www.kernel. org.

[10] http://www.quantum.com/Solution s/datadeduplication/ Index.aspx.

.

[11] Dmsetup - low level logical volume management. Manual Page. Dmsetup.

[12] Red Hat Inc. Device-mapper Resource Page. Website. http://sources.redhat. com/dm/.

[13] dm-crypt: a device-mapper crypto target for Linux. Website. http://www.saout.de/misc/dm-crypt/