DEXT3: Block Level Inline Deduplication for EXT3 File System

Amar More
M.A.E. Alandi, Pune, India
ahmore@comp.maepune.ac.in

Zishan Shaikh
M.A.E. Alandi, Pune, India
zishan366shaikh@gmail.com

Vishal Salve
M.A.E. Alandi, Pune, India
vishaluttamsalve@gmail.com

Abstract

Deduplication is basically an intelligent storage and compression technique that avoids saving redundant data onto the disk. Solid State Disk (SSD) media have gained popularity these days owing to their low power demands, resistance to natural shocks and vibrations and a high quality random access performance. However, these media come with limitations such as high cost, small capacity and a limited erase-write cycle lifespan. Inline deduplication helps alleviate these problems by avoiding redundant writes to the disk and making efficient use of disk space. In this paper, a block level inline deduplication layer for EXT3 file system named the DEXT3 layer is proposed. This layer identifies the possibility of writing redundant data to the disk by maintaining an in-core metadata structure of the previously written data. The metadata structure is made persistent to the disk, ensuring that the deduplication process does not crumble owing to a system shutdown or reboot. The DEXT3 layer also takes care of the modification and the deletion a file whose blocks have been referred by other files, which otherwise would have created data loss issues for the referred files.

1 Introduction

While data redundancy was once an acceptable part of the operational backup process, the rapid growth of digital content storage has made organizations approach this issue with a new thought process and look for other ways to optimize storage utilization. Data deduplication would make disk more affordable by avoiding backing up redundant data.

Data deduplication is basically a single-instance storage method which helps in reducing the storage needs by eliminating redundant data. Only one instance of the data is actually stored and the redundant data, which will not exist physically, is just a pointer to the unique data. For example consider an example of an email system which may contain 100 instances of a 10 MB attachment. If the entire system was to be backed up, all 100 instances would be saved requiring 1000 MB of disk space. With deduplication in place, only 10 MB of one instance would actually exist and the other redundant instances would just be a pointer to this unique instance saving precious disk space.

Data deduplication generally works at the file or block level. The latter outperforms the former because file level deduplication would work for files that are same as a whole; whereas block level deduplication would work for blocks (a constituting part of the file) as a whole. Each chunk or block of data, that is about to be written to the disk, is subjected to hash algorithm such as MD5. This process generates a unique hash value for each block, which is stored in a database for further referral. If a file is updated, then only the added or changed data is stored, and disk storage is allocated only for these parts. In our work, we present an implementation of an inline block level deduplication layer added to the EXT3 file system, named as the DEXT3 layer.

2 Related Work

2.1 Types of Deduplication

Deduplication has various implementation approaches which can be significantly classified by their resiliency level (file vs. block) or by when they perform deduplication (inline vs. post-process) or by their method of duplicate data detection (hash vs. byte wise comparison). As stated earlier, inline deduplication is a strategy in which the duplicate data is identified before it hits the disk. Post-process deduplication is a strategy in which the data is first written to the disk and then de-duplication processing occurs in the background. In hash-based strategies, the process uses cryptographic hashes to identify duplicate data whereas byte wise strategies compare the data itself directly.

2.2 Deduplication Targets

Venti [1] and Foundation [2] both perform deduplication with respect to a fixed block size. Venti is basically an archival system. Foundation is a system that stores snapshots of various virtual machines and also uses Bloom filters to detect potential duplicates on the system.

The NetApp deduplication function [3] for file servers is used in integration with WAFL and FlexVol [4] and makes use of hashes to find duplicate data blocks. In this system, hash collisions are resolved through byte by byte comparison. This process runs in the background, therefore making it a post-process deduplication system.

The LBFS [5], Data Domain [6], HYDRAstor [7], REBL [8] and TAPER [9] identify and detect potential duplicates using content-defined chunks.

2.3 Deduplication Performance and Resources

The Data Domain [6] is a deduplication system that makes use of the spatial locality of data in a given backup stream to improve the performance of searching for hash metadata of the same data stream.

Sparse indexing [10] is a technique of deduplication that reduces the size of the data information kept in the RAM. It achieves this by sampling data hashes of data chunks.

These processes work fine with large data sets in a provided locality; however, we have to assume that the access patterns have small or no locality in a primary storage system.

2.4 Performance With Solid State Disks (SSD)

It is generally efficient to use fast devices like Solid State Disks for frequently used data. Chunkstash [12] and dedupv1 [11] make use of solid state disks for metadata. SSD and metadata is a good combination, mainly because I/O operations for metadata are always small and random.

3 DEXT3 design

The proposed design works on the following two principles:

1. Provide a working file system which tries to save space and avoid redundant disk writes.
2. Organize the in-core data structure efficiently and make it persistent to the disk.

In order to find the potential duplicates, the write system call invoked by the kernel is intercepted in the VFS itself. The data that is about to be written to the disk is available in a buffer in the write system call. This buffer is then broken into chunks of 4 kB each, owing to the 4 kB block size design of the kernel. These 4 kB blocks are compared with the data that has been previously written. If the new data block matches any previously written block, then instead of writing out the new block, the file's metadata is updated to point to the existing block on disk.

In order to identify potential duplicate blocks efficiently, an in-memory mapping of data hashes and the corresponding block numbers of the data is maintained. This mapping is created whenever there is a successful write to the disk. When control stays within the VFS, the hash value of block data is inserted into the data structure, and when the file system allocates a block to this data, its block number is stored in the data structure. In order maintain correct file system behaviour, another field is added in the data structure which maintains the reference count of the block, which indicates how many times a particular block has been used. This count is used whenever the kernel is about to release a block.

Deduplication could be implemented in a block layer device driver, which sits in between the file system layer and the actual underlying block device. This approach has the advantage that it is general and can be used for any file system. However, this extra layer of indirection incurs a performance penalty. The proposed implementation simply uses the existing block pointers in file system metadata, regardless of whether the pointer points to a regular block or a deduplicated block.

The system may be shut down or rebooted at any time. Being held in memory, the entire data structure would be lost, and after the system restarts again, the kernel would be unaware of the deduplicated blocks. Deleting a file whose blocks have been deduplicated would cause data loss issues to the referring files. The user might modify the file whose blocks have been referred by other files. This will still cause issues to the files which point to the blocks contained by the file being modified. All these issues would be handled in the proposed design.

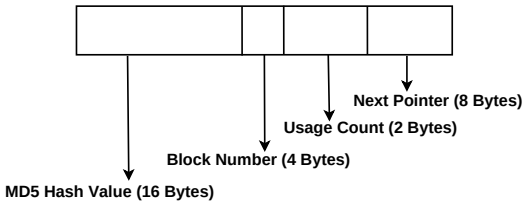

Figure 1: Node Structure of Dedupe Database

4 Implementation Details

DEXT3, a version of the EXT3 filesystem is implemented for Linux to provide on-the-fly inline block-level deduplication. DEXT3 is written for the Linux kernel version 2.6.35. The occurrence of duplicate data is detected in the write system call itself and then controls block allocation when the control of the kernel flows to the file system. The block de-allocation process of the file system is also modified in order to avoid freeing deduplicated blocks. The coming sections explain the design in detail.

4.1 Dedupe Database and Working of Deduplication Mechanism

The dedupe database is basically a data structure that maintains metadata with respect to the chunk of data that was previously written to the disk. The data structure that has been implemented is hash table with linear chaining. The size of the table is static, but the size of the chain is dynamic and grows as entries are added to the structure.

Figure 1 shows the structure of a node in this chain. The node occupies a total 30 bytes in size. The size of this node is fixed. The fields of this node describe the data chunk written to the disk in detail, that is, its hash value, the block number allocated by the kernel and the usage count which shows how many times the block with this block number has been referred by files. This field is useful in handling block de-allocation.

Following two hash functions are used by DEXT3:

MD5 Message Digest 5

FNV Fowler Nollvo hash

MD5 is used to generate hash value of the 4 kB data chunk which is intercepted in the write system call. This

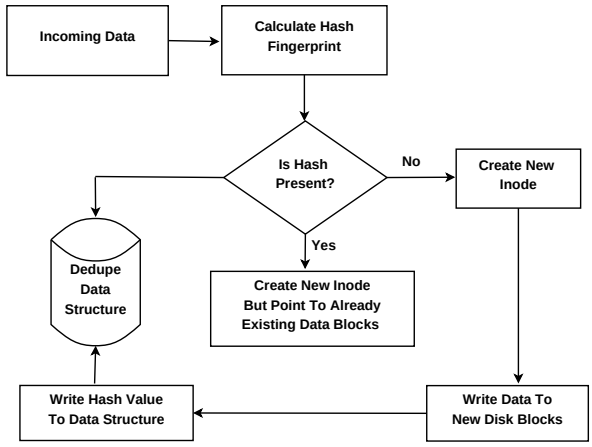

Figure 2: Working of the DEXT3 layer

hash value is then used for checking redundant and duplicate data in the write system call.

The FNV hash is used to build the data structure itself. This function returns an integer hash value of 32-bits. This value is further truncated to 21-bits to keep the memory requirements optimal. So with a 21-bit FNV hash, a total 2097152 indices could be used to build the table. At every index one linear chain is maintained. The structure of a node in this chain is as stated in Figure 1.

First the 16-byte hash value of the 4 kB data chunk is obtained. Then the same chunk is subjected to FNV hash to obtain an index in the linear chain. The MD5 hash is inserted in this chain, and at a later stage when the kernel allocates a block to this data chunk, the newly allocated block number is stored in the same node and the usage count of this node is initialized to 1.

Before inserting the hash value in the chain, the chain lookup is performed. If the lookup fails, then the data is new and therefore not redundant, and the kernel is allowed to follow its normal allocation routines. However if the lookup succeeds, then the data is redundant. The usage count of the matching node is incremented, and the kernel does not allocate a new block for this data. This entire process is carried out before the data chunk is actually allocated a block on to the disk. Figure 2 explains the above stated design.

4.2 Main File Deletion and Modification

The deletion of a file whose blocks are referred by other files would cause data loss issues. To prevent this unlikely event, another data structure called the *character*

bitmap is introduced. This bitmap is a character array that maintains the deduplicated status of all the blocks available in the file system. The bitmap status of a block is set to deduplicated (i.e. 1), whenever we end up performing a successful lookup in the linear chain for that block number. It would imply that the block has now been referred by more than one files. The status of a block remains 0 in the bitmap if it has never been deduplicated. The bitmap is efficient in terms of both memory and lookup. Each byte in the array can hold the status of eight blocks at once. Looking up the status of a block involves a single logical AND operation.

Before the kernel goes deeper into file deletion, the DEXT3 layer first decrements the usage count of all the blocks held by the target file. If the usage count of a block reaches zero, it means that the file about to be deleted is the last file to ever refer this block. So the layer:

1. Deletes its node from the chain
2. Updates its bitmap status to zero

If the usage count does not reach to zero, the DEXT3 layer simply allows the kernel to proceed to the next step.

The next step in file deletion is releasing the blocks held by the file. When the kernel is about to de-allocate and release a specific block, the DEXT3 layer first checks the deduplicated status of that block in the bitmap. If the status returned is 1, it means that there are files in the file system those still refer to this data block and releasing this block would be problematic. In this case the DEXT3 layer does not allow the kernel to release this block. Owing to this strategy, even if the main file inode has been deleted, if it holds any deduplicated block, then those blocks would still be available for use by the deduplicated files. Figure 3 and 4 explain this design.

The next challenge is to handle the modification of a file whose blocks have been deduplicated. Once again, this would cause data loss issues to the deduplicated blocks. Whenever a file is modified, such as via an editor, the kernel is asked to do the following things:

1. De-allocate the current inode
2. Allocate a new inode

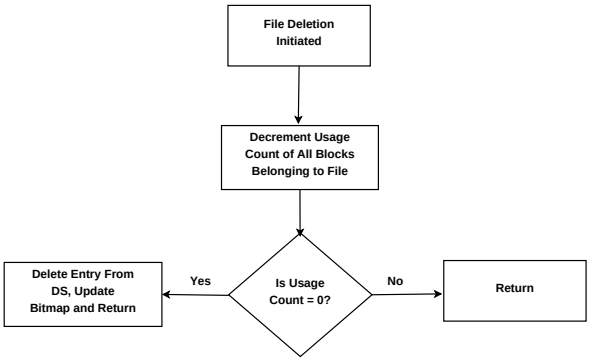

Figure 3: File Deletion Phase - I

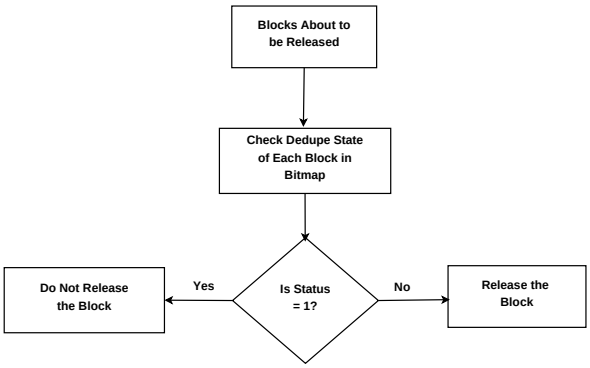

Figure 4: File Deletion Phase - II

When the kernel deletes the current inode, it de-allocates the blocks that are currently held by the file. When a block is about to be allocated, the control is as shown in Figures 3 and 4.

When the kernel allocates a new inode to the file, the control flows through the write system call again. As the DEXT3 layer exists in the write system call, the control flow is as shown in Figure 2.

4.3 Permanent Data Structure

The entire DEXT3 database resides in the memory and is therefore volatile. Whenever the system is shutdown or rebooted, this highly precious information is lost. After the system boots again, the kernel is unaware of which block has been deduplicated. If it were about to release a block, then it would have directly released that block, even it was deduplicated earlier. To prevent this catastrophic event, the data structure is made persistent to the disk. When the system boots and the deduplication process starts again, data structure is rebuilt from this saved information. The bitmap is not flushed to the disk. Instead when the data structure is rebuilt, at the

% of Deduplication	Partition Capacity (GB)	Saved Disk Space (GB)
0	136	0
10	150	14
20	164	28
30	177	41
40	190	54
50	204	68
100	272	136

Table 1: Statistics with respect to the data structure

% of Deduplicated Blocks	Total Blocks Saved (GB)	Saved Disk Space (MB)
10	104961	410
20	209922	820
30	314884	1230
40	419845	1640
50	524806	2050
100	104961	4100

Table 2: Statistics with respect to the File Size

same time the bitmap is updated. This strategy keeps the kernel informed about the deduplicated status of each block even if the system is booted any number of times. This also helps to maintain and start the deduplication process from the point where it had stopped due to system shutdown.

5 Statistics for Disk Space Saving using DEXT3

5.1 Statistics With Respect to the Data Structure

Considering the number of nodes in a linear chain to be 17, we calculate the following statistics: With 17 nodes present at each of the 2097152 indices in the table, the size of the data structure goes up to nearly 1 GB. With this 1 GB of metadata, the layer manages 136 GB of data stored on to the disk, without any possibility of deduplication.

In the entire structure, if 10% of blocks have been deduplicated, then the partition capacity rises virtually to 150 GB saving nearly 14 GB of disk space. With 100% deduplication, partition capacity doubles to 272 GB, saving complete 136 GB disk space.

5.2 Statistics With Respect to File Size

Considering the size of a file to be 4 GB, the total number of blocks that would be allocated to this file would be 1049612.

6 Conclusion

The statistics and results were encouraging and indicate how DEXT3 is able to save significant amount of disk space and reduce the number of disk writes. The memory requirements remain optimal. Including deduplication in the kernel introduces a little overhead in the system performance. Though on one hand we introduce this overhead, on the other hand DEXT3 provides significant cost, space and energy savings. We view this as acceptable since performance gain is not the primary goal, rather our goal is to avoid writes and achieve space savings. The DEXT3 layer does not modify any other filesystem metadata other than the inode. The inode is updated at run time itself, so there is no need to update the file system metadata explicitly. The persistent data structure strategy makes it possible to rebuild the data structure after system boot. Almost all the applications that use solid state disks for primary storage make use of the existing and standard file systems. Providing a simple DEXT3 layer is crucial in order to promote real world use. The original FFS in UNIX added disk awareness to an otherwise hardware oblivious filesystem. We find block level inline deduplication i.e. DEXT3 as a crucial and important layer of SSD limitations. More importantly, as the industry transitions away from the spinning disks towards solid state devices, this kind of approach, as we see, will become increasingly critical.

References

[1] S. Quinlan and S. Dorward, *Venti: a new approach to archival storage*. The First USENIX conference on File and Storage Technologies (Fast '02), January 2002.

[2] S. Rhea, R. Cox and A. Pesterev, *Fast, inexpensive content-addressed storage in Foundation*. The 2008 USENIX Annual Technical Conference, June 2008.

[3] C. Alvarez, *NetApp deduplication for FAS and V-Series deployment and implementation guide*. Technical ReportTR-3505, January 2010.

[4] S. Rhea, R. Cox and A. Pesterev, *Fast, inexpensive content-addressed storage in Foundation*. The 2008 USENIX Annual Technical Conference, June 2008.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, *A low-bandwidth network file system*. The 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Alberta, Canada, October 2001.

[6] B. Zhu, K.Li, and H. Patterson, *Avoiding the disk bottleneck in the Data Domain deduplication file system*. The 6th USENIX Conference on File and Storage Technologies (FAST '08), February 2008.

[7] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, *HYDRAstor: a scalable secondary storage*. The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[8] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey, *Redundancy elimination within large collections of files*. The 2004 Usenix Annual Technical Conference, June-July 2004.

[9] N. Jain, M. Dahlin, and R. Tewari, *TAPER: tiered approach for eliminating redundancy in replica synchronization*. The 4th USENIX Conference on File and Storage Technologies (FAST '05), December 2005

[10] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise and P. Camble, *Sparse indexing: large scale, inline deduplication using sampling and locality*. The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[11] D. Meister and A. Brinkmann, *dedupv1: improving deduplication throughput using solid state drives (SSD)*. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), May 2010.

[12] B. Debnath, S. Sengupta and J. Li, *ChunkStash: speeding up inline storage deduplication using flash memory*. The 2010 USENIX Annual Technical Conference, June 2010.

[13] Keren Jin, Ethan L. Miller, *The Effectiveness of Deduplication on Virtual Machine Disk Images*. SYSTOR 2009 May 2009, Haifa, Israel.