Hammerspace : Data deduplication for btree based Linux file systems

Chinmay Kamat
Pune Institute of Computer Technology
chinmaykamat@gmail.com

Gaurav Tungatkar
Pune Institute of Computer Technology
gauravstt@gmail.com

Kushal Dalmia
Pune Institute of Computer Technology
kdpict@gmail.com

Amey Magar
Pune Institute of Computer Technology
amey.magar@gmail.com

Abstract

Continually growing storage farms, shrinking backup windows, increasing demand for nearly instantaneous data restores and increasing operational costs are the consequences of the increasing digital data. The increasing amount of data stored and backed up in data centres is a major concern. A keen observation is that most backup jobs only hold a small percentage of really new data – typically less than 5%. The rest is a duplicate of data that has remained unmodified from the previous backup. The elimination of this duplicate data promises to reduce storage needs and improve data restore times considerably.

Data deduplication is a method of reducing storage needs by eliminating redundant data. Only one unique instance of the data is actually retained on storage media. Each subsequent instance is just referenced back to the one saved copy. This paper proposes a block level data deduplication mechanism for btree based Linux file systems. It describes a design which includes a Fingerprint Index and a Locality Based Bucket mechanism. The Locality Based Bucket Layout and Fingerprint Index enable fast and efficient detection and elimination of duplicate data blocks. The design is integrated into the file system and does not require any application level intelligence.

We have built a prototype of the system in the new Tux3 file system and present some preliminary results.

Index Terms – Deduplication, Buckets, Fingerprints, Tux3.

1 Introduction

Today's data centers deal with enormous volumes of data and have massive storage requirements. These increasing storage needs cause a serious problem. Typically, data centers perform a weekly full backup of all the data on their primary storage systems to secondary storage devices where they keep these backups for weeks to months. In addition, they may perform daily incremental backups that copy only the data which has changed since the last backup. For disaster management, data will be replicated at off-site centers. For recovery, the data is transferred from off-site over a wide area network. The network bandwidth requirement for this will be enormous.

Most of these systems generate huge amounts of redundant data as most of the files remain unchanged with respect to the previous backup. Disk based deduplication saves disk space by eliminating redundant duplicate data for such storage systems, thus allowing storage of much more data on a given amount of disk space [1, 2]. This paper describes the design and implementation of efficient data deduplication using an on-disk btree based Fingerprint Index and a Locality Based Bucket Layout. The Locality Based Bucket Layout acts as a cache and prevents the bottleneck that is caused due to continuous look up of on-disk index [3].

We present results based on a prototype with the Tux3 file system. Tux3 is a new age write anywhere, atomic commit file system which scales well for large amounts of storage. This scalability makes Tux3 a very good choice for archival and backup systems where deduplication is most required.

2 Data Deduplication

2.1 Concept

In the context of disk storage, a deduplication algorithm searches for duplicate data objects, such as blocks, chunks, or files, and discards these duplicates. When a duplicate object is detected, its reference pointers are modified so that the object can still be located and retrieved, but it "shares" its physical location with other identical objects. This data sharing is the foundation of all forms of data deduplication.

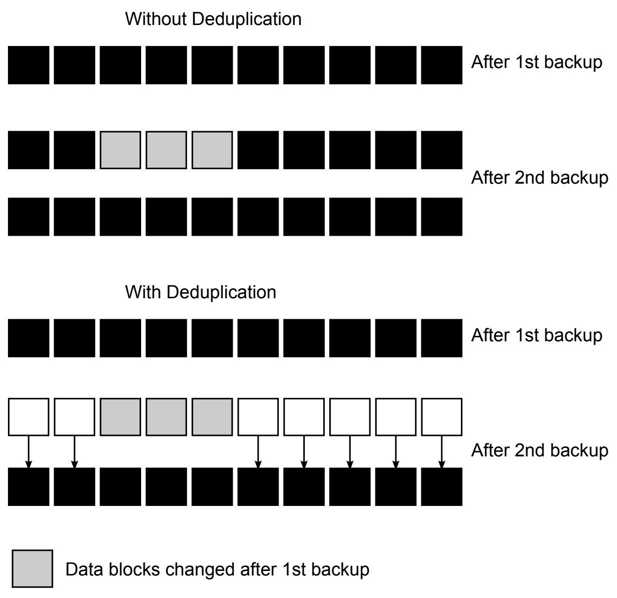

Figure 1: Concept of Data Deduplication

Deduplication technology analyzes the incoming backup data and stores each unique block of data only once, using pointers for additional instances of the same data. When duplicate data is found, a pointer is established to the original set of data as opposed to actually storing the duplicate blocks – removing or "deduplicating" the redundant blocks from the volume.

2.2 In-line v/s Post processing

In-line deduplication sometimes referred to as synchronous deduplication, involve analyzing the data as it is being written. If duplicates are found, the data is not written to the disk or transmitted over the network. The advantage is that the data does not have a big landing footprint; landing footprint being amount of disk space/network bandwidth the data takes the first time it is written. However, since analyzing duplicates can be computationally intensive, this can limit the rate at which data is written to a deduplication enabled system. Post processing or asynchronous deduplication methods attempt at duplicate elimination after data is written to the disk. Data is first written to the disk as is. It is later analyzed by a daemon process independent of the write process, to find duplicates. This daemon then modifies existing references and meta data to eliminate duplicate data. The advantage is that write speed is not hampered. However, this method has several complexities. Consider a backup of 1 GB that need to go on a disk having 700 MB space. Say after de-duplication it is reduced to 500 MB. Since its landing footprint is 1 GB, post processing based appliances cannot write this backup to this disk even though it has enough space for the compressed backup. There always has to be provision for the maximum footprint. Other concerns include modifying the meta-data by the daemon and how it is handled for live data that may be read or written to in the midst of such a modification.

2.3 Benefits

Data deduplication provides a cost-effective way of packing more data on a single disk. Thus disks fill up at slower rate allowing retaining backup data on disk for longer time. Lost or corrupt files can be quickly and easily restored from backups taken over a longer time span. This makes file restores fast and easy from multiple recovery points. Efficient use of disk-space reduces the cost-per-gigabyte of storage by requiring fewer disks than in conventional backup systems. This can considerably reduce floor space and energy consumption. This leads to significant saving in operational costs. Data deduplication also makes the replication of backup data over low-bandwidth WAN links viable. Only the changed data can be sent over the network.

Many current implementations work at the application level. This makes the deduplication technique dependant on the application. Moreover, these solutions work on structured data making them ineffective for unstructured data.

Moving deduplication to the file system layer makes it effective even for unstructured data. This implementation is application agnostic thereby making it easier to implement it in many environments. It can work with any backup software allowing quick and cheaper deployment than application level solutions.

3 Tux3 Linux file system

Tux3 is an open-source versioning file system created by Daniel Phillips [4]. In broad outline, Tux3 is a conventional Unix-style inode/file/directory design with wrinkles. A Tux3 inode table is a B-tree with versioned attributes at the leaves. A file is an inode attribute that is a B-tree with versioned extents at the leaves. Directory indexes are mapped into directory file blocks as with H-Tree. Free space is mapped by a B-tree with extents at the leaves.

Introduced on	23rd July, 2008
Directory Contents	B-tree
File Allocation	B-tree
Max file size	2^{60} bytes (1 EB)
Max number of files	2^{48}
Max volume size	2^{60} bytes (1 EB)
Default block size	4 KB

Table 1: The Tux3 Linux file system

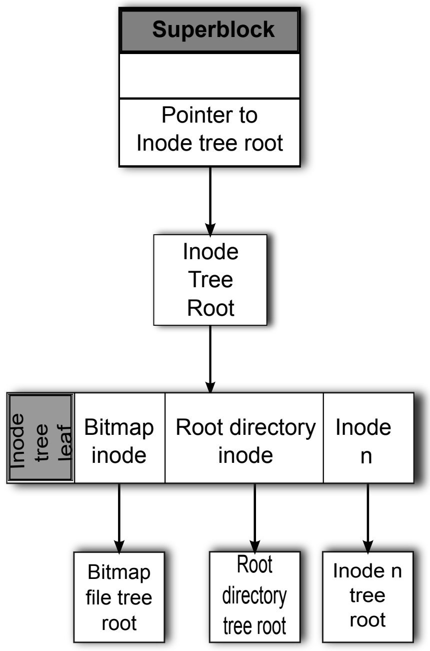

Figure 2: Tux3 file system structure

4 Design

The implementation of data deduplication in Tux3 does not change the file system semantics and user applications need not be aware of the underlying deduplication. The design includes an on-disk btree called Fingerprint Index and buckets which are used as locality caches.

4.1 Fingerprints and Fingerprint Index

The deduplication is performed at file system data block level. Unique data blocks are identified by the hash of their contents. By using a collision resistant hash function with a sufficiently large output it is possible to consider the hash of the data block as unique. Such a unique hash is called the fingerprint of a block.

For the choice of hash function, a cryptographic hash function is chosen because it is computationally infeasible to find two distinct inputs that hash to the same value. A cryptographic hash function can also be used to ensure file system integrity. We use SHA-1 to calculate the fingerprint.

SHA-1 is a popular hash algorithm for security systems. Assuming random hash values with a uniform distribution, a collection of n different data blocks and a hash function that generates b bits, the probability p that there will be one or more collisions is bounded by the number of pairs of blocks multiplied by the probability that a given pair will collide [1] , i.e.

$$p \leq \frac{n(n-1)}{2} \times \frac{1}{2^b}$$

Today, a large storage system may contain data in petabytes (2^{50} bytes) or an exabyte (2^{60} bytes) stored as 4 Kbyte blocks (2^{48} blocks). Using the SHA-1 hash function, the probability of a collision is less than 2^{-64} (approximately 10^{-20}) [1]. Such a scenario seems sufficiently unlikely that we ignore it and use the SHA-1 hash as a unique identifier for a block.

The fingerprint index is a btree based data structure. A B-tree of order n is a multi-way search tree with two properties: All leaves are at the same level and the number of keys in any node lies between n and 2n, with the possible exception of the root which may have fewer keys. A btree node contains n block pointers separated by n-1 keys. Searching a balanced tree means that all leaves are at the same depth. There is no runaway pointer overhead. Indeed, even very large B-trees can guarantee only a small number of nodes must be retrieved to find a given key. For example, a B-tree of 10,000,000 keys with 50 keys per node never needs to retrieve more than 4 nodes to find any key. This property makes btree an ideal choice for the fingerprint index.

The root of the fingerprint index is maintained in the file system superblock. The fingerprint btree is indexed by `hashkeys` - the first 64 bits of the SHA-1 fingerprint of the data of the block being written to the file system. It serves as a look-up structure and helps in quick and efficient detection of duplicate data blocks. Each node in the index tree contains multiple `<hashkey, block no.>` pairs where the block numbers point to nodes in the next tree level.

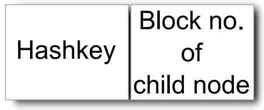

Figure 3: Fingerprint Index node entry

The penultimate level nodes have block numbers which point to index tree leaves. The index tree leaves contain multiple triples of `<hashkey, block no. of bucket, offset>`. The `offset` is used to directly access the relevant entry rather than traversing all the bucket entries.

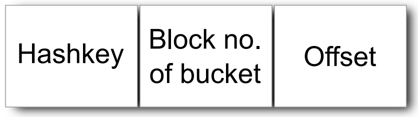

Figure 4: Fingerprint Index leaf node entry

4.2 Locality Based Buckets

As shown by previous research, data locality can be exploited to accelerating duplicate detection process [3]. A keen observation of backup data reveals that such locality exists and blocks tend to reappear in the same or very similar sequences across multiple backups.

Consider a normal file composed of a hundred or more 4KB blocks. Every time that file is backed up, the same sequence of a hundred blocks will appear. If the file is modified slightly, there will be some new blocks, but the rest will appear in the same order. When new data contains a duplicate block x, there is a high probability that other blocks in its locale are duplicates of the neighbors of x.

This observation can be used to prevent excessive disk I/O and improve performance by the introduction of the bucket abstraction. A bucket is a mapping between the SHA-1 hash values (fingerprints) and the corresponding block numbers. The buckets are filled contiguously such that the mapping of blocks belonging to the same locality are most often stored in the same bucket. The bucket entries store the entire 160 bit SHA-1 hash value.

For implementation, the in-memory inode of the files in the file system maintain two block numbers – the *current read bucket* and the *current write bucket*. The current read bucket is a bucket which has a high probability of containing entries for blocks belonging to the same locality as the block being written. The current write bucket is the bucket to which the new entries of the blocks being written are made and therefore entries corresponding to neighboring blocks are together. The buckets are 4KB data blocks containing entries in the form `<fingerprint, block no., reference count>`.

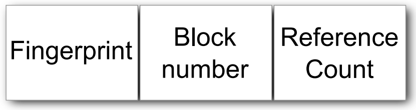

Figure 5: Entry in the bucket

The reference count is incremented each time a duplicate block is detected and its reference added to the file system structure. The reference count is used to make sure that the block is not freed if it is being referred by multiple files. The buckets also allow for a very novel approach to manage hashkey collisions by keeping all the colliding entries together in a single ‘collision bucket’.

5 Algorithm

To implement in-line deduplication, changes need to be made to the write mechanism of the file system. The algorithm for the same is as follows -

Input – Buffer containing data to be written

```
1. Compute the SHA-1 hash of the buffer data
2. Perform a hash look-up in the current read bucket
3. If found,
     - Do not allocate a new physical block.
     - Obtain duplicate block number from the read bucket entry.
     - Use this block number in the file's data leaf node.
     - Increment reference count of corresponding bucket entry.
4. Else,
     - Obtain the first 64-bit of the SHA-1 fingerprint.
     - Perform look-up in the fingerprint index
     - If found,
         i. Read the bucket number from the fingerprint index entry.
        ii. Read the data block of the bucket into memory and make it the current read bucket.
       iii. Compare the full 160-bit SHA-1 hash of the buffer with the corresponding bucket entry.
        iv. If match not found, handle collision.
         v. Obtain the corresponding physical block number from the bucket entry.
        vi. Use this block number in the file's data leaf node.
       vii. Increment reference count of corresponding bucket entry.
     - Else,
         i. Allocate a new physical block.
        ii. Add a new entry into the fingerprint index and the current write bucket.
       iii. Perform a disk write of the block.
```

To elaborate the working of deduplication, consider an example scenario -
A file *x* containing 10 data blocks is to be written. None of these blocks have same content as any of the existing file system blocks on disk. Hence, fingerprints of all these blocks would not match any index entry. This would result in allocation of 10 new data blocks and their corresponding entries being made into the fingerprint index and the current write bucket say *b*. Consider another file *y* resulting from appending some data to *x*. Let this result into *y* containing 14 data blocks of which first 10 remain same as *x*. For the write of the first block, fingerprint match will be found in the index and no new physical block will be allocated. The bucket containing the corresponding entry - *b* would replace the current read bucket. For the subsequent 9 duplicate blocks, the look-up would be performed only in the bucket *b* which is already present in memory. This avoids the need to traverse the on-disk fingerprint index significantly reducing disk I/O. As no new physical blocks are needed for these duplicate blocks too, the deduplication mechanism will greatly improve system performance and save disk space. For the last 4 blocks for which no duplicates exist the procedure is similar to the one followed for *x*. Thus a 14 block file *y* required the allocation and disk writes of only 4 data blocks.

6 Performance Analysis

Our current implementation is a part of the Tux3 FUSE (File System in Userspace) port. The figures mentioned represent the implementation of deduplication in userspace and have been computed on a Intel Core 2 Duo, T5450, 1.66GHz with 1GB 667MHz DDR2 RAM using a 10 GB partition on a Hitachi HTS54251 160 GB, 5400 rpm hard disk.

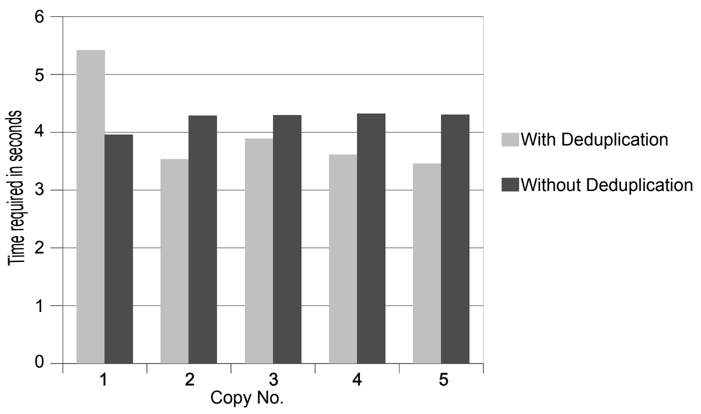

Figure 6: Archival performance - Copying same file multiple times

The above graph represents the timing profile for copying a 100MB video file repeatedly with and without deduplication. As observed, with deduplication enabled, the time taken for the first copy is more than that without deduplication. But for successive copies, deduplication reduces time required significantly. Thus, the Locality Based Bucket Layout enables fast duplicate detection and copying which offsets any performance hit observed in the first copy. Also, there is considerable space savings by avoiding disk blocks for the 4 duplicate copies. The above process which consumes 500MB in a normal file system, would use $\sim$ 100MB after deduplication.

Database copies	Without dedup (No. of blocks allocated)	With dedup (No. of blocks allocated)	% Saving
1	64223	59787	6.90
2	128434	63789	50.33
3	192645	67798	64.80

Table 2: Performance analysis using real world data

The above table shows the number of 4KB blocks required for copying the MySQL test employee database containing 3,00,024 entries with and without deduplication. The 1st copy is the original database. The 2nd copy is the database with changes to 20,000 random entries with random values. The 3rd copy is the database with 20,000 random changes over the 2nd copy. These values include the meta-data blocks along with the actual data blocks. It can be observed that there is considerable amount of space saving, approximately 50-55% for the entire process.

7 Other Applications

Besides archival storage, deduplication can be used in the following applications –

1. Email Servers

   In the case of an e-mail server, deduplicated storage would mean that a single copy of a message is held within its database whilst individual mailboxes access the content through a reference pointer. Apart from the benefit of reduction in disk space requirements, the primary benefit is to greatly enhance delivery efficiency of messages sent to large distribution lists.

2. Storage systems of multimedia sharing websites

   With the advent of Web 2.0, the popularity of websites which allow multimedia sharing has grown manifold. Without content based deduplication, these websites would have to store multiple redundant copies of the same multimedia file uploaded by various users. Deduplication would remove this redundancy and allow more distinct content to be uploaded onto such servers.

3. Common store for Virtual machine images

8 Advantages

1. Significantly Less Storage
2. Better Performance
3. Provision for data integrity check using the fingerprint index

9 Conclusion

Data deduplication technology represents one of the most significant storage enhancements in recent years. Deduplication offers the ability to store more on a given amount of storage and enables replication using low-bandwidth links, both of which improve cost effectiveness.

In this paper, we present a block level deduplication design suitable for archival storage and backup. The btree based fingerprint index, indexes the data blocks by the fingerprints of their content. It serves as a structure for fingerprint look-up and accelerates the process of duplicate detection. The Locality Based Bucket Layout introduces the bucket abstraction which reduces excessive disk I/O and improves the deduplication performance considerably for a continuous run of duplicate blocks. Both these concepts when implemented together provide an effective and fast deduplication mechanism. We implement data deduplication in the btree based Tux3 file system which scales well for huge amounts of data and therefore proves to be an ideal choice for all kinds of data backups and archival storage. The in-line deduplication implementation is transparent for the system user as no changes are made to the user level semantics of duplicate files.

Apart from backup systems data deduplication would also prove useful in applications such as email servers, storage systems of multimedia sharing websites database and file system integrity check.

The deduplication design presented in this paper satisfies all the criteria of a good space saving solution. It does not compromise data integrity and reliability, operates seamlessly in all user environments and has minimal impact on system performance .

10 Acknowledgements

We wish to thank the following people for their contributions to the paper; Mr. Amey Inamdar, our mentor whose ideas and timely criticism helped us bring this to fruition. Mr. Daniel Phillips, creator Tux3, for his continued support and guidance. Prof. Rekha Kulkarni, our faculty guide at PICT. Last, but not the least, we wish to thank the anonymous reviewers, who patiently read our paper and gave us valuable inputs which helped make significant improvements in this paper.

References

[1] S. Quinlan, S. Dorward, *Venti: a new approach to archival storage* (2002) [Online] Available: `http://plan9.bell-labs.com/sys/doc/venti/venti.pdf`

[2] P. Kulkarni, F. Douglis, J. LaVoie, J. M. Tracey, *Redundancy Elimination Within Large Collections of Files* (2004) [Online] Available: `http://www.usenix.org/event/usenix04/tech/general/full_papers/kulkarni/kulkarni.pdf`

[3] B. Zhu, K. Li, H. Patterson, *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System* (2008) [Online] Available: `http://www.usenix.org/event/fast08/tech/full_papers/zhu/zhu.pdf`

[4] Daniel Phillips, *The Tux3 Linux File System* (2008) [Online] Available: `http://mailman.tux3.org/pipermail/tux3/2008-July/000005.html`