Space Savings and Design Considerations in Variable Length Deduplication

Giridhar Appaji Nag Yasa
NetApp Inc
giridhar@netapp.com

P. C. Nagesh
NetApp Inc
nageshc@netapp.com

ABSTRACT

Explosion of data growth and duplication of data in enterprises has led to the deployment of a variety of deduplication technologies. However not all deduplication technologies serve the needs of every workload. Most prior research in deduplication concentrates on fixed block size (or variable block size at a fixed block boundary) deduplication which provides sub-optimal space efficiency in workloads where the duplicate data is not block aligned. Workloads also differ in the nature of operations and their priorities thereby affecting the choice of the right flavor of deduplication. Object workloads for instance, hold multiple versions of archived documents that have a high degree of duplicate data. They are also write-once read-many in nature and follow a whole object GET, PUT and DELETE model and would be better served by a deduplication strategy that takes care of non-block aligned changes to data.

In this paper, we describe and evaluate a hybrid of a variable length and block based deduplication that is hierarchical in nature. We are motivated by the following insights from real world data: (a) object workload applications do not do in-place modification of data and hence new versions of objects are written again as a whole (b) significant amount of data among different versions of the same object is shareable but the changes are usually not block aligned. While the second point is the basis for variable length technique, both the above insights motivate our hierarchical deduplication strategy.

We show through experiments with production data-sets from enterprise environments that this provides up to twice the space savings compared to a fixed block deduplication.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; E.2 [**Data**]: DATA STORAGE REPRESENTATIONS—*Object representation*; E.4 [**Data**]: DATA STORAGE REPRESENTATIONS—*Data compaction and compression*

General Terms

Design, Algorithms, Experimentation, Measurement

Keywords

Deduplication, Space Efficiency

1. INTRODUCTION

Explosive growth of data storage has been a well witnessed phenomenon in recent times [1, 2]. Data retention is also growing due to compliance needs and prospective benefits from data mining and analytics. However, not all data growth is comprised of unique data as several copies of the same files are stored by multiple users and several versions of the same documents are stored that share lots of common data. In enterprise archives, several full data backups are stored that contain a significant amount of duplicate data. Therefore, more and more enterprises are seeking capacity optimized storage solutions such as deduplication that enables storing a larger logical data set than what the underlying physical disk space can hold.

Deduplication is a popular storage efficiency trend that originated primarily for backup and archival workloads, but has increasingly found acceptance in primary workloads. In its simplest form, data deduplication is known as single instance storage [3] (or SIS), wherein one or more files that are identical in content are replaced with logical references pointing to a single copy of the file on the disk. For example, with SIS, a PowerPoint file sent to several users over email is stored as a single physical copy and there are logical references to this copy from user's mailbox directories. Therefore the size of the logical data set is much larger than the actual physical disk space required for storing it.

Single instance storage fails to benefit from multiple files which differ by small amounts. An example is an image library that contains duplicate header data across all the files. Multiple versions of the same file too contain significant amount of duplicate data that cannot be shared by SIS. Block level deduplication [4, 5] addresses this issue by identifying duplicate filesystem blocks and gives better space savings. In block level deduplication, the unit of sharing is a filesystem block. This enables data sharing across similar but different files, whenever there are duplicate disk blocks. Most filesystems are fixed block size file systems, where the disk is divided into fixed size (usually 4KB) blocks and files are represented as a sequence of pointers to these blocks.

In several environments, duplicate data exists, but may not be aligned to the block structure of the fixed block size filesystem. One of the causes of unaligned duplicate data is the *data shifting problem*, where in the data stream is shifted from its original block structure on account of minor insertions or deletions. This effect is often found in Microsoft® PowerPoint/Word documents and also in full backup datasets by IBM Tivoli Storage Manager (identical data that is backed up from two directories with names of different lengths suffers the data shifting problem). Variable length deduplication addresses this issue by choosing block boundaries in a context dependent manner [6, 7, 8] and therefore shown to give greater space savings than fixed block deduplication.

A key issue concerning deduplication (fixed size or variable length) is the filesystem metadata overheads. Deduplication requires that data blocks (of fixed size or variable length) be referred to by logical pointers from multiple files. The unit of data sharing is often fixed at a small value (4KB) and duplicate data detection is performed by segmenting the whole file at these small sized blocks. This approach is inefficient when large contiguous sequences of data exist, as multiple logical pointers have to be maintained where a single logical reference should ideally have sufficed. Also, directly detecting duplicate data at the largest possible granularity saves the overheads of processing mutiple small sized blocks and improves deduplication efficiency.

The following insights have shaped our solution in this work:

- Object store and archival workloads do not involve in-place modification of data and new versions of objects are written in their entirety. These environments (write once read many) facilitate us to optimize on space efficiency without worrying about complexities of handling data modification.

- Document repositories have a high degree of duplicate data on account of storing multiple versions of documents. But this is unaligned duplicate data and variable length deduplication is required to optimize on space efficiency.

We are primarily motivated by the observation that large contiguous sequences of unaligned duplicate data exists in document repositories and backup datasets. Detection and sharing of duplicate data should be done at the largest possible granularity to optimize deduplication and filesystem metadata overheads. A hierarchical approach of data processing has been studied in other contexts such as WAN optimization ([9, 10]) with the goal of detecting differing data at the smallest possible granularity. Similar techniques can be utilized to detect the duplicate data at the largest possible granularity for storage efficiency.

In this paper we describe the design and evaluation of a hybrid of a variable length and block based deduplication that is hierarchical in nature.

Important contributions of this paper are:

- Demonstrating the increased storage space efficiency (sometimes by over a factor of two) of variable length deduplication over a fixed size block based deduplication on real world datasets.

- Design of a hierarchical approach to deduplication and demonstrating its efficacy on real world data sets. We show that a significant part (minimum of 35% to maximum of 98%) of duplicate data is found at larger (64KB or more) sized blocks.

- Design of a constrained form of variable length deduplication to meet the needs of a performance sensitive multiple-block extent based file system (section 5.5). We have modified the variable length deduplication to identify extent boundaries that are on multiples of 64(or 128 etc.) bytes only. We demonstrate through experimental results that this constrained form of variable length deduplication provides nearly the same amount of space savings as an unconstrained variable length deduplication.

The rest of this paper is organized as follows: We compare and contrast with related work in section 2. We implemented the hierarchical variable length deduplication algorithms described in section 3, and the design and implementation are detailed in section 4. We demonstrate the increased space savings of variable length deduplication as compared to a fixed block based deduplication with real world datasets in section 5. We conclude in section 6 and suggest enhancements to this work for future in section 7.

2. RELATED WORK

Deduplication techniques for storage space efficiency employ different techniques (e.g. different granularity, content awareness etc.) to optimize for a variety of workloads (e.g. backup and primary environments) and data characteristics.

Granularity: Space efficiency has been achieved by deduplicating at whole-file granularity (e.g. SIS [3] or [11], in the context of copy detection by Brin et. al [6] and finding similar files by Manber [12]) or more commonly by identifying duplicate data at block level (e.g. Venti [4], ZFS [5], EMC Centera [11], HP [13] and NEC HYDRAstor [14]). In our work, we employ a hierarchical variable length deduplication, where duplicate data segments are detected at the largest possible granularity.

Content awareness: Some techniques use content awareness to achieve better space efficiency like in Ocarina [15] where the block boundary and size is based on the knowledge of the file layout of popular file types (compression is tightly integrated with deduplication in their solution). HP [13] offers different technologies for different business segments e.g. object-level differencing for archival workloads via special delta files. In contrast, our solution is content and format agnostic and is not restricted to one or few file types.

Throughput considerations: To achieve good throughput, Zhu et. al. [7] use bloom filters, Lillibridge et. al [16] use sampling and locality, and Xia et. al [17] use similarity and locality. Srinivasan et. al. [18] exploit spatial and temporal locality for inline deduplication for primary storage. In certain solutions, throughput is maintained during ingest by doing offline deduplication (e.g. NetApp® in [19] and [20] which also compare data to assure its integrity). We are focused on object stores and WORM (write once read many) workloads and our primary goal is maximizing space savings.

Chunking and hierarchies: Hash trees or Merkle trees [21] have been used in peer-to-peer systems [22] to scale torrent file size with content, and for data transfers between computers by Hamilton et. al [10] to speed up backup windows by minimizing the network traffic. Muthitacharoen et. al. [9] also use deduplication techniques to enable running of network file systems over slow or wide-area networks. The effects of various chunking approaches have been studied by Meister et. al. in [23]. We employ similar techniques but are focussed on disk space savings and aimed to find the suitability of hierarchical variable length deduplication for primary and backup workloads.

In summary, our work focuses on maximizing physical space savings, it is content and format agnostic (all data patterns are assumed to be equally likely and does not need updating for future new types of file formats). The techniques are suitable for offline deduplication for object workloads [24] and backup streams, and whole object inline deduplication for small objects. The techniques are also extendable to offline deduplication of primary data.

3. OUR APPROACH

The crux of the variable length deduplication lies in how data is segmented. It differs from fixed size data segmentation in that it is designed to alleviate the data shift problem. Essentially, this is the first step in variable length deduplication, referred to as context specific segmentation. This is described in detail the section 3.1

An important characteristic of archival data is that there are segments of duplicate data of different lengths. In the simplest case, we could look for duplicate data segments of 4KB or whole object duplicates corresponding to several GBs, or at an intermediate granularity of few MBs. In our work, we organize the data at multiple levels of granularity (three in our prototype), with the goal of identifying duplicate data at as large a granularity as possible. These levels are organized into a hierarchical tree. This is detailed in section 3.2.

Finally, we describe our hierarchical deduplication mechanism in section 3.3, which is essentially a progressive search for duplicate data, starting at highest levels of granularity (object level) and ending at the lowest (4KB blocks).

3.1 Context Specific Segmentation

In a fixed size block filesystem the smallest addressable data segments are the fixed size file blocks which are obtained by equal sized chunking of the logical space of the file data. Therefore the chunk boundary offsets are always integral multiples of the block size. In contrast, content specific segmentation chooses segment boundaries based on specific patterns occurring in the data, which is identified by a rolling hash. This is described in Algorithm 1.

```
input  : Data[0:N] A file or data stream of length N
output: Synchronization Points[]
A set of synchronization points that are offsets in
the range [0..N-1]
for $i \leftarrow 0$ to $N-1$ do
    value ← rolling hash of Data[$i-w, i$] ;
    if $value \oplus MASK = 0$ then append i to
    synchronization points ;
end
```
Algorithm 1: Context Specific Segmentation

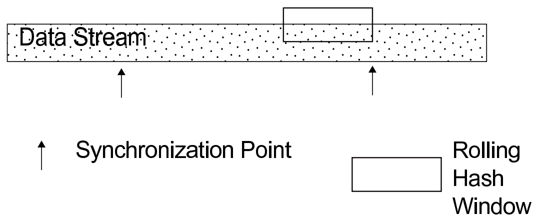


Figure 1: Synchronization point identification with a rolling hash window

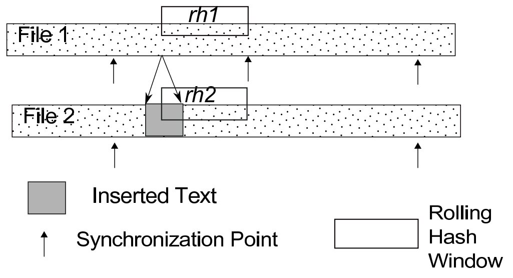


Figure 2: Synchronization points in a data stream that has a minor insertion

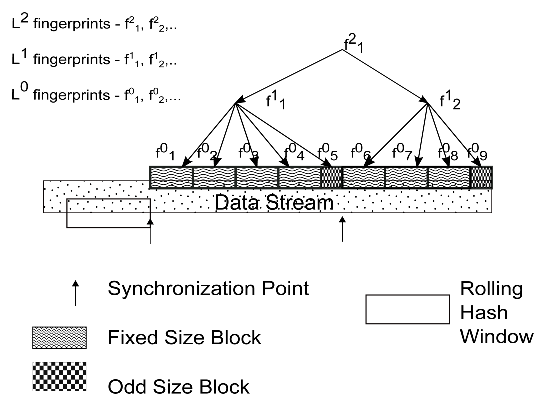
We employ a cyclic polynomial [25] based implementation for computing rolling hash [26]. The rolling hash value is checked for a matching pattern and regard that as a synchronization point if the pattern matches. The pattern search is a check for M specific bit positions in the hash value. When no knowledge of data is assumed, all possible values of the rolling hash are equally probable and therefore on an average, one out of every 2^M rolling hash values obtained at step (a) will satisfy the data pattern. This translates to an average length of 2^M bytes between two synchronization points. We can control the average chunk length by altering a $MASK$ parameter and in this implementation we have fixed it to yield an average chunk length of 64KB.

To see how this alleviates the unaligned duplication problem, consider the example in Figure 2. File 1 and File 2 contain identical data except for minor insertion at the beginning of File 2.

As illustrated earlier in Figure 2, the rolling hash values for the two files would be identical up to the point of insertion in the new file. As the rolling window straddles over the newly inserted bytes in File 2, it produces different set of rolling hash values as compared to the original File 1. Finally as the rolling window fully leaves the inserted portion, it encounters the same byte stream as in the original file, producing the same set of rolling hash values as the original file. Therefore, all subsequent synchronization points in the original file are identified at corresponding positions in the new file, at offsets shifted by a constant width, equal to that of the newly inserted bytes. The data between these set of synchronization points in the two files are identical.

We may observe that a modification to a file affects at most one segment, preserving the other segment boundaries. Therefore the loss of deduplication due to data shift problem is restricted to at most one segment, unlike in fixed block size deduplication where hash values of all blocks after the insertion are different from the hash values of the blocks in the original data.

3.2 Hierarchical Fingerprinting



L^2 fingerprints - f^2_1, f^2_2,..
L^1 fingerprints - f^1_1, f^1_2,..
L^0 fingerprints - f^0_1, f^0_2,...
f^2_1
f^1_1
f^1_2
f^0_1 f^0_2 f^0_3 f^0_4 f^0_5 f^0_6 f^0_7 f^0_8 f^0_9
Data Stream
Synchronization Point
Fixed Size Block
Odd Size Block
Rolling Hash Window

Figure 3: Merkle Tree of Fingerprints

We employ a hybrid of variable and fixed length segmentation. We identify synchronization points as described above (section 3.1), which are spaced apart by around 64KB on an average. This is then divided into a sequence of 4KB fixed size blocks, with potentially one odd sized block at the end. To summarize, a file is divided into segments whose boundaries are decided by the Algorithm 1 above. These segments are subdivided into blocks which are mostly 4KB in size and some few odd sized blocks that are less than 4KB. We then compute a hierarchical tree of fingerprints over this block structure as shown in Figure 3.

The algorithm for hierarchical fingerprinting is shown in Algorithm 2

3.3 Hierarchical Deduplication

In a fixed block deduplication system such as Venti [4] or ZFS [5], there is a single level of fingerprints corresponding to the filesystem blocks. In hierarchical deduplication, there are multiple (three in our implementation) levels of fingerprints corresponding to data segments of different lengths. A fingerprint at a higher level represents multiple fingerprints contained in the subtree rooted at it in the tree of fingerprints (Figure 3). Deduplication proceeds in a hierarchical fashion by starting at the highest level (L2) and proceeding all the way down to the lowest (L0).

When duplicates are detected at a higher level, processing of all fingerprints that are within its subtree is skipped, therefore reducing the overall deduplication time. For e.g. if duplicate fingerprints are found at the object (L2) level, whole objects are implied to be identical and fingerprints corresponding to the recipient objects at the lower levels (L1 and L0) are removed. In this case, a single object is retained as the donor and the remaining object references (recipients) are made to logically point to this donor.

This procedure ensures that duplicate regions of data are detected at the highest possible granularity. The process of deduplication within a level is accomplished by sorting the fingerprint database in fingerprint order, leading to duplicates ending up at the same position.

4. IMPLEMENTATION

We implemented identification of duplicate data using fixed size block deduplication algorithm and our variable length deduplication algorithms as described in this paper on a machine with 8 GB RAM, 2 dual-core 1.8 GHz AMD Opteron

```
input : D, F
D is the current block of data of file ;
output: L0, L1, L2
3 lists of fingerprints corresponding to 3 levels i.e block level, segment level and object level fingerprints.;
$L2 \leftarrow \emptyset$  $L1 \leftarrow \emptyset$  $L0 \leftarrow \emptyset$;
$B \leftarrow \emptyset$ B is an internal buffer;
$B \leftarrow B + D$ append D to the buffer;
$S[0 \dots N] \leftarrow$ Context Specific Segmentation($B$) ;
find synchronization points in $B$;
$LS \leftarrow [(S[0], S[1]), (S[1], S[2]) \dots (S[N-1], S[N])]$
foreach $(s, e) \in LS$ do
    LS is the list of segments. (s,e) are the segment boundaries given by consecutive synchronization points ;
    $L1Data \leftarrow \emptyset$;
    $LB \leftarrow [(s, s + 4k - 1), \dots (s + mk, e)]$;
    LB is the list of blocks within the segment (s,e)
    foreach $b \in LB$ do
        $fp^0 \leftarrow$MD5 hash of block b ;
        $L0 \leftarrow L0 \cup fp^0$ ;
        $L1Data \leftarrow L1Data + fp^0$ ;
        concatenate the string $fp^0$ with L1Data ;
    end
    $fp^1 \leftarrow$ MD5 hash of L1Data ;
    $L1 \leftarrow L1 \cup fp^1$ ;
    $L2Data \leftarrow L2Data + fp^1$ ;
    L2Data is the concatenation of L1 fingerprints of that object
end
$fp^2 \leftarrow$ MD5 hash of L2Data ;
$B \leftarrow B[S[N] :]$ trim the buffer, up to the last synchronization point
```
Algorithm 2: Hierarchical Fingerprinting

CPUs with 14 7200 RPM SATA drives attached. This machine was running RedHat Linux® 9 (kernel 2.4.8-20smp).

The important parameters for the fixed block size deduplication implementation were a block size of 4KB and a MD5 [27] hash to identify each data block.

For the variable length deduplication, we used a 32 bit custom rolling hash with a window of 4KB, MD5 for the block hashes and three levels in a Merkle tree per object. The rolling checksum, MD5 hash computation and creation of (or appending to) the hash database were done in-band. A sufficiently strong hash function (other than MD5) may be used with similar results.

Both the implementations used an in-house external memory merge-sort implementation to batch process identification of duplicate data.

5. EVALUATION AND RESULTS

The goals of our experimental evaluation is to measure the space savings of variable length deduplication, utility of hierarchical deduplication and finally the suitability of variable length deduplication in a constrained extent based file system. Since space savings are data set dependent, we have included data sets from a wide category of workloads. We tested for space savings with our techniques and compared it against the space savings from a fixed block size deduplication that we implemented (with the block size being 4KB). A break up of the distribution of duplicate data across the three levels of the hierarchy is shown to establish the value of hierarchical deduplication. Finally, we show how our solution can beneficially impact deduplication for an extent based file system design, towards better storage efficiency.

5.1 Experimental Data Sets

Our experiments were focussed on two important classes of workloads, viz.: primary object store workloads and backup workloads. We have obtained real life production data from a live enterprise, that represent these workloads. A summary of these is presented in Table 1.

	Attributes		
Dataset	Workload type	Size	Description
Debian ISOs	Primary	272GB	Debian ISO images
SharePoint	Primary	30GB	PDF, PPT files
Oracle	Primary	125GB	Raw database dump
NBU Home	Backup	150GB	Backup images
NBU Oracle	Backup	135GB	Backup images
TSM Oracle	Backup	140GB	Backup images

Table 1: Datasets used for experiments

We crawled a live enterprise SharePoint® repository to obtain a collection of documents (PDFs, Microsoft Office Word, PowerPoint and Excel files). This (SharePoint) is a 30GB dataset consisting of about 9000 small files, where each file is generally a few MBs in size.

We also downloaded weekly generated CD ISO images from Debian repository [28], corresponding to weekly releases spanning around two months. This (Debian ISOs) is 272GB dataset of about 300 files where each file is 600MB to 700MB in size.

The above two datasets are representative of primary object store workloads. We also show the results on a raw data dump of a live enterprise database (Oracle®) that represents a primary workload.

We also collected backup images of Engineering Home, Mail and Database(Oracle) dump from two popular backup applications NBU (Symantec™ NetBackup™ [29]) and IBM TSM (Tivoli Store Manager [30]). These are three nightly full backups of fast changing data (spread apart a few days) and are 120GB to 150GB each. These datasets are representative of archival or backup workloads.

5.2 Space Savings Results

The space savings obtained by variable length deduplication for the above datasets is depicted and compared with that obtained by fixed size block deduplication in Figures 4 and 4. We can see that variable length deduplication performs as good as or even better than fixed block size deduplication for most datasets.

In the case of the primary datasets Debian ISOs and Oracle, the space savings is around 15% more when compared to fixed block size deduplication as shown in Figure 4). With SharePoint data, the space savings is more than twice (Figure 4).

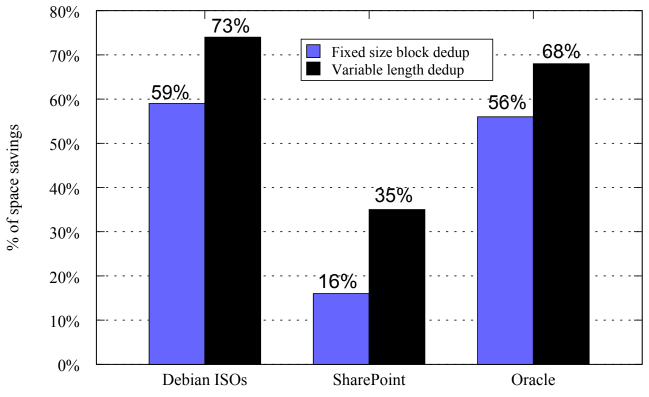


Figure 4: Percentage of physical space saved across various primary data sets

We tested our algorithms against a few other data-sets (1.2 TB in overall size) as well (e.g. The Ensemble Genome database, data files from the OpenArena, Warzone2100, Wesnoth and Widelands games, and Unigene database). However, the amount of duplicate data in all these cases is either too low or too high and the percentage of space savings difference in these data sets obtainable by fixed size block deduplication and variable length deduplication is negligible. This again supports our claim that variable length deduplication is at least as good as fixed size block deduplication.

With images of backup streams, the space savings obtained with variable length deduplication was slightly better for NBU Home images and TSM Home images (better by 3% to 5%) but more than twice in case of NBU Oracle images as shown in Figure 5.

5.3 Hierarchical Space Savings

We analyzed the spread of duplication across the three levels of blocks (L0), segments (L1) and whole object (L2) for these datasets and this is illustrated in Figure 6 for primary data, and and in Figure 7 for backup data streams.

Note that a large contiguous duplicate data is represented by a single logical reference pointer. For example a large

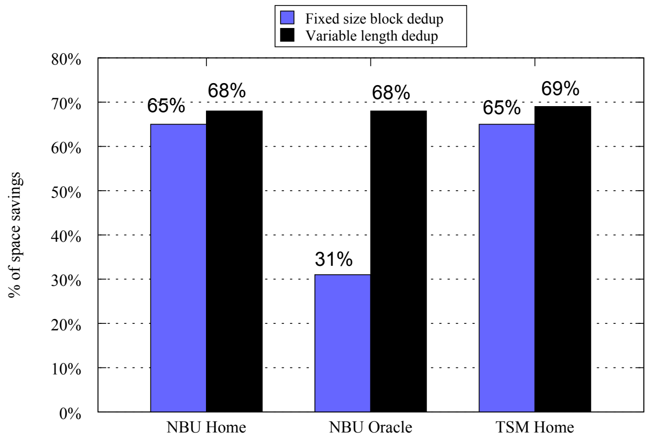


Figure 5: Percentage of physical space saved: backup streams

wholly duplicate object requires a single pointer in contrast to having multiple pointers for each of its constituent blocks, therefore reducing metadata overhead. The importance and benefits of hierarchical deduplication is seen across these datasets, where significant amount of duplicate data is found at higher levels. By finding duplicates at higher levels, we skip the processing of many fingerprints under its subtree, therefore greatly reducing the overall deduplication time.

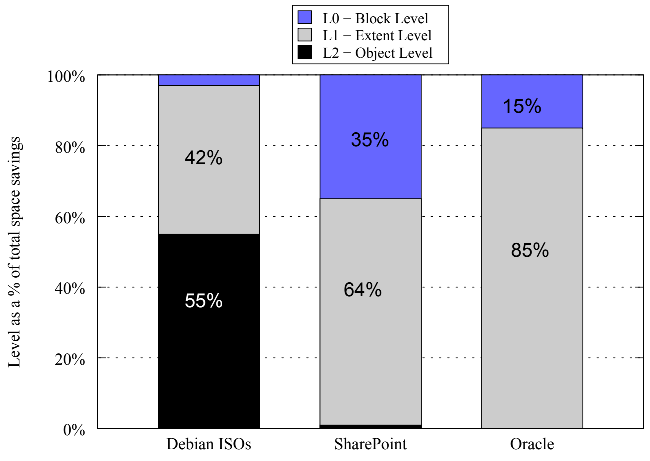


Figure 6: Contribution of different levels in hierarchical deduplication to space saved across various primary data sets

5.4 Deduplication metadata overheads

While hierarchical deduplication offers space savings in the filesystem metadata, the deduplication meta data size increases marginally as we now have multiple fingerprint databases corresponding to each levels. The size of L0 FPDB[1] depends on the total amount of the data in the volume, as is the case with fixed size block deduplication. Not accounting for the odd sized blocks, this translates to one fingerprint per $4KB$ of data. Around 16 block (L0) fingerprints combine to form one segment (L1) fingerprint as the average segment length is $64KB$ in our solution. Therefore, the L1 FPDB is around $\frac{1}{16}$ the size of the L0 FPDB. The size of L2 FPDB is solely dependent on the number of objects in the volume. In general for a hierarchical tree of fingerprints, where N fingerprints at each level combine to produce one fingerprint at the higher level, the size of the FPDB decreases by a factor of $\frac{1}{N}$ as we move up the tree. Therefore, the total size of all the FPDBs combined has an upper bound of $\frac{N}{N-1}$ times the size of the L0 FPDB.

[1]FPDB: Fingerprint Database

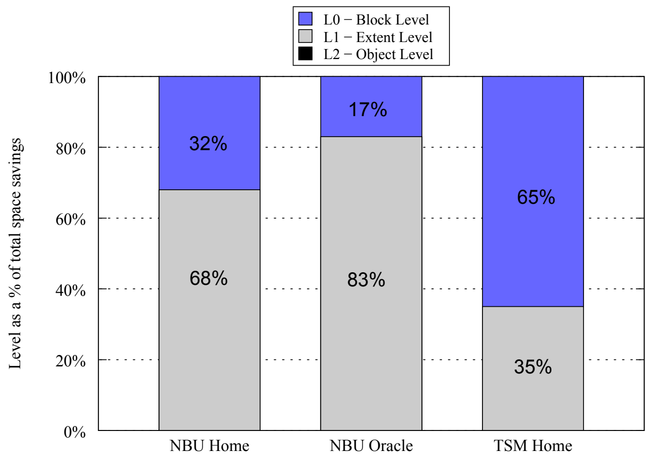


Figure 7: Contribution of different levels in hierarchical deduplication to space saved across various backup stream images

In hierarchical deduplication, lower level FPDBs are filtered or pruned whenever duplicates are found at higher level FPDBs. Therefore, the overall deduplication time is reduced. This advantage is lost in cases where no duplicates can be found at higher level FPDBs. In this case, the total processing time of all the FPDBs combined has an upper bound of $\frac{N}{N-1}$ times the time required for processing just the L0 FPDB.

5.5 Relevance to Extent Based Filesystems

Extent based file systems are usually not aligned to an arbitrary byte and have some constraints on how the extents may be formed. It is expected that extents boundaries can only be at multiples of 64, 128 or 256 byte-offsets. To suit this requirement, we modified context specific segmentation algorithm, such that synchronization points are rolled over to their nearest 64 byte multiples. We refer to this as 64 byte aligned variable length deduplication.

With this change, we re-evaluated the space savings over the same datasets and the results are quite encouraging. Either there is a marginal loss of space savings (less than 3%) or 64 byte aligned deduplication gives results that are as good as from byte aligned deduplication. This is plotted in Figure 8 for primary data, and in Figure 9 for backup data streams.

It is important to note that the data shifting problem would continue to exist in this case if the insertions or deletions in data are not in multiples of 64 bytes. However, this may be solved by aligning byte aligned synchronization points in the files to the extent boundary in the file system (as indicated in Section 6) and forgoing space savings in the odd sized blocks.

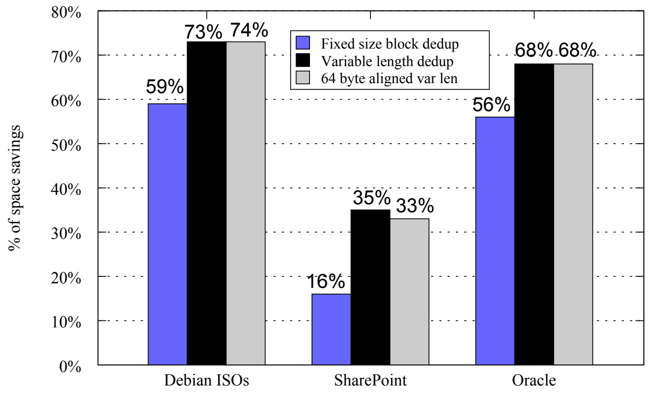


Figure 8: Fixed size block vs. variable length vs. 64 byte aligned variable length deduplication space savings: primary data sets

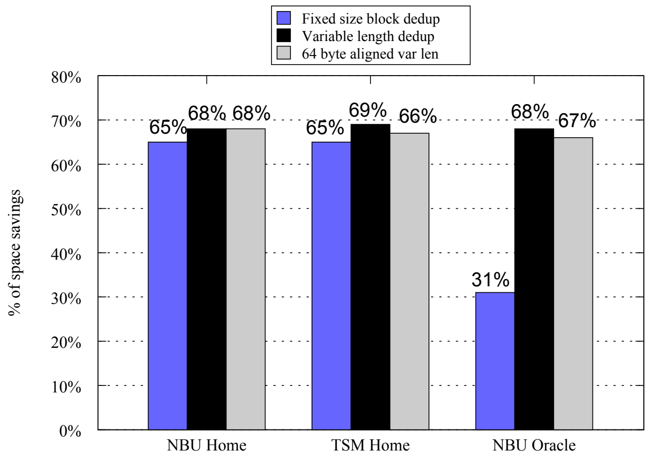


Figure 9: Fixed size block vs. variable length vs. 64 byte aligned variable length deduplication space savings: backup data streams

6. CONCLUSION

To summarize, our main contribution is in establishing the value of variable length deduplication by incorporating novel design modifications. We have tested our implementation over real world datasets and have shown that variable length deduplication gives much greater space savings than fixed block size deduplication and is sometimes better by over a factor of two. We have also experimented with a constrained form of variable length deduplication, where the extent boundaries are restricted to be multiples of 64 bytes. We have shown that variable length deduplication is still effective even in this constrained form and therefore offers a practical alternative for performance sensitive extent based filesystems.

One disadvantage of variable length deduplication is that, handling data modifications is now more complex. The variable length segment, in which data is modified, will have to be processed again to recompute the L0 fingerprints. Further, these changes will have to be propagated up to L1 and L2, i.e. through all the levels of the Merkle tree, up to the root. Therefore, variable length deduplication is ideally suited for WORM (Write Once Read Many) workloads and object stores that do not support object or file modification. These environments would then benefit from increased space savings without compromising on other performance metrics.

In a file system that already has fixed block based deduplication implemented, the rolling hash based context specific segmentation may be used to layout data so that block boundaries are aligned to the synchronization points. Additional meta-data would be required to identify and handle odd-sized blocks and a bit of space is wasted because of internal fragmentation but our results show that the overall space savings is significant to warrant such a change.

7. FUTURE WORK

In order to extend the described techniques to file-systems that allow file data to be overwritten (and not just replaced with a new version of the file), the rolling hash values may be stored on disk alongside the actual data to support incremental updates to the Merkle tree. Some file-systems employ block checksums for reliability (e.g. ZFS [5] and BtrFS [31]), and using the rolling hash value at the block boundary as a substitute for that purpose would support incremental updates and also decrease the overall computation in the system.

Additional space savings may be achieved by computing additional block level (L0) checksums backwards from the synchronization points. This would give slightly better space savings in case of insertions and deletions by restricting the loss of space savings to less than a segment.

Our solution is based on three levels of hierarchical deduplication. However, one might increase the number of levels when dealing with larger object sizes to capture contiguous data at different levels of granularity. More levels might be formed by creating a binary tree beginning from L1 upwards.

Alternately, a rolling hash based mechanism might again be employed over the data formed by concatenating the segment fingerprints, to decide the next level of segmentation. This process could then be recursively applied till we reach the whole object level.

For smaller objects, deduplication at the object level can be performed inline. The index required to quickly lookup object level hashes would be significantly smaller and should enable this either by storing all of it in RAM or in SSDs [32].

8. REFERENCES

[1] Laura DuBois and Marshall Amaldas. Key considerations as dedup lication evolves into primary storage. *IDC*, 2010.

[2] John Gantz and David Reinsel. The 2011 digital universe study: Extracting value from chaos. *IDC*, 2011.

[3] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single instance storage in windows 2000. *Proceedings of the 4th USENIX Windows systems symposium*, 2000.

[4] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. *Proceedings of the 1st USENIX conference on file and storage technologies*, 2002.

[5] Jeff Bonwick. ZFS deduplication. *https://blogs.oracle.com/bonwick/entry/zfs%20dedup*, 2009.

[6] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy detection mechanisms for digital documents. *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, 1995.

[7] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. *Proceedings of the 7th USENIX conference on file and storage technologies*, 2008.

[8] Quantum. Data deduplication background: A technical white paper. *http://www.quantum.com/iqdoc/doc.aspx?id=5959.*

[9] Athicha Muthiacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[10] Dr. Jim Hamilton and Eric W. Olsen. Design and implementation of a storage repository using commonality factoring. *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[11] Terri McClure and Brian Garrett. EMC centera, optimizing archive efficiency. *White Paper*, 2009.

[12] Udi Manber. Finding similar files in a large file system. *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[13] Hewlett Packard. Understanding the HP deduplication strategy. *http://www.usdatavault.com/library/understanding deduplication.pdf.*

[14] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. *Proccedings of the 7th USENIX conference on file and storage technologies*, 2009.

[15] George Crump. Lab report: Deduplication of primary storage. *http://www.ocarinanetworks.com/products/products-overview*, 2009.

[16] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. *Proccedings of the 7th USENIX conference on File and storage technologies*, 2009.

[17] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

[18] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[19] Carlos Alvarez. NetApp deduplication for FAS and V-Series. deployment and implementation guide. *NetApp TR-3505*, 2009.

[20] Keith Brown. Deduplicating backup data streams with the NetApp VTL. *NetApp Inc.*, 2009.

[21] Ralph. C. Merkle. A digital signature based on a conventional encryption function. *Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, 1987.

[22] Arvid Norberg. Merkle hash torrent extension. *www.bittorrent.org/beps/bep 0030.html*, 2009.

[23] Dirk Meister and Andre Brinkmann. Multi-level comparison of data deduplication in a backup scenario. *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[24] SNIA Technical Position. Cloud Data Management Interface (CDMI) Version 1.0.2. *SNIA*, 2012.

[25] Jonathan D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems (TOIS), Volume 15 Issue 3*, 1997.

[26] Michael O. Rabin. Fingerprinting by random polynomials. *TR-15-81*, 1981.

[27] Ronald L. Rivest. The MD5 message-digest algorithm. *IETF RFC 1321*, 1992.

[28] The Debian Project. Weekly builds. *http://cdimage.debian.org/cdimage/weekly-builds/.*

[29] Symantec. Netbackup. *http://www.symantec.com/netbackup.*

[30] IBM. Tivoli storage manager. *http://www.ibm.com/software/tivoli/products/storage-mgr/.*

[31] Oracle Corporation. Btrfs checksum tree. *http://en.wikipedia.org/wiki/Btrfs.*

[32] Dirk Meister and Andre Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.