Duplicate Data Elimination in a SAN File System

Bo Hong
Univ. of California, Santa Cruz
hongbo@cs.ucsc.edu

Demyn Plantenberg
IBM Almaden Research Center
demyn@almaden.ibm.com

Darrell D.E. Long
Univ. of California, Santa Cruz
darrell@cs.ucsc.edu

Miriam Sivan-Zimet
IBM Almaden Research Center
mzimet@us.ibm.com

Abstract

Duplicate Data Elimination (DDE) is our method for identifying and coalescing identical data blocks in Storage Tank, a SAN file system. On-line file systems pose a unique set of performance and implementation challenges for this feature. Existing techniques, which are used to improve both storage and network utilization, do not satisfy these constraints. Our design employs a combination of content hashing, copy-on-write, and lazy updates to achieve its functional and performance goals. DDE executes primarily as a background process. The design also builds on Storage Tank's FlashCopy function to ease implementation.[1]

We include an analysis of selected real-world data sets that is aimed at demonstrating the space-saving potential of coalescing duplicate data. Our results show that DDE can reduce storage consumption by up to 80% in some application environments. The analysis explores several additional features, such as the impact of varying file block size and the contribution of whole file duplication to the net savings.

1. Introduction

Duplicate data can occupy a substantial portion of a storage system. Often the duplication is intentional: files are copied for safe keeping or for historical records. Just as often, the duplicate data appear through independent channels: individuals who save the same email attachments or who download the same files from the web. It seems intuitive that addressing all of this unrecognized redundancy could result in storage resources being used more efficiently.

Our research goal is to reduce the amount of duplicated data in on-line file systems without significantly impacting system performance. This performance requirement is what differentiates our approach, which we call Duplicate Data Elimination (DDE), from those used in backup and archival storage systems [1, 6, 23]. To minimize its performance impact, DDE executes primarily as a background process that operates in a lazy, best-effort fashion whenever possible. Data is written to the file system as usual, and then some time later, background threads find duplicates and coalesce them to save storage. DDE is transparent to users. It is also flexible enough to be enabled and disabled on an existing file system without disrupting its operation, and flexible enough to be used on parts of the file system, such as select directories or particular file types.

Duplicate Data Elimination (DDE) is designed for IBM Storage Tank [17], a heterogeneous, scalable SAN file system. In Storage Tank, file system clients coordinate their actions through meta-data servers, but access the storage devices directly without involving servers in the data path. DDE uses three key techniques to address its design goals: content-based hashing, copy-on-write (COW), and lazy update. DDE detects duplicate data on the logical block level by comparing hashes of the block contents; it guarantees consistency between block contents and their hashes by using copy-on-write. Data is coalesced by changing corresponding file block allocation maps. COW and lazy update allow us to update the file block allocation maps without revoking the file's data locks. Together these techniques minimize DDE's performance impact.

Figure 1 shows an example of coalescing duplicate data blocks in an on-line file system. Before coalescing, files F_1, F_2 and F_3 consume 11 blocks. However, they each contain a common piece of data that is three blocks in size. Clients are unaware of this duplication when they write these files. The server detects the common data later and coalesces the identical blocks. After coalescing, F_1–F_3 consume only five blocks in total by sharing the same three blocks. Six blocks are saved, resulting in a 55% storage reduction.

[1]Storage Tank technology is available today in the IBM Total Storage SAN File System (SANFS). However, this paper and research is based on underlying Storage Tank technology and may not become part of the IBM TotalStorage SAN File System product.

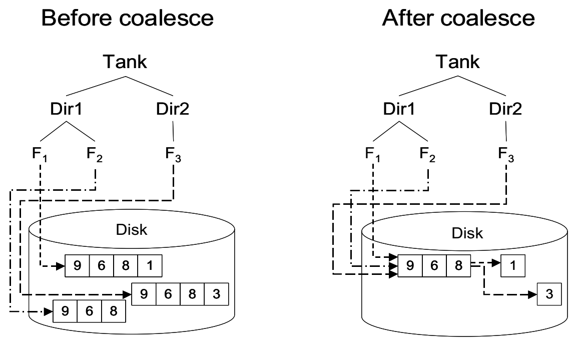


Figure 1. An example of coalescing duplicate data blocks.

2. Background

IBM Storage Tank is a multi-platform, scalable file system that works with storage area networks (SANs) [17]. In Storage Tank, data is stored on devices that can be directly accessed through a SAN, while meta-data is managed separately by one or more specialized Storage Tank meta-data servers. Storage Tank clients are designed to direct all metadata operations to Storage Tank servers and to direct all data operations to storage devices. Storage Tank servers are not involved in the data path.

The current version of Storage Tank works with ordinary block-addressable storage devices such as disk drives and RAID systems. The basic I/O operation unit in Storage Tank is a block. The storage devices are required to have no more intelligence than the ability to read and write blocks from the volumes (LUNs) they present. Storage Tank file data is also managed in block units. The size of a file block is typically a multiple of the device block size.

Storage Tank exposes three new abstractions called *file sets*, *storage pools*, and *arenas*. These are in addition to the traditional abstractions found in file systems such as files, directories, and volumes. A file set is a subtree of the global namespace. It groups a set of Storage Tank files and directories for the purpose of load balancing and management. A storage pool is a collection of one or more volumes. It provides a logical grouping of the volumes for the allocation of space to file sets. A file set can cross multiple storage pools. An arena provides the mapping between a file set and a storage pool. As such, there is one arena for each file set that has files in a particular storage pool. The arena abstraction is strictly internal to the Storage Tank server, but is an important element in duplicate data elimination. Using an arena, Storage Tank can track the used and free space owned by a file set in a storage pool, and specifies the logical to physical mapping of space in the file set to the volumes in the storage pool.

The Storage Tank Protocol provides a rich locking scheme that enables file sharing among Storage Tank clients or, when necessary, allows clients to have exclusive accesses to files [5, 7, 8]. A Storage Tank server grants locks to clients, and the lock granularity is per file. There are three file lock modes in Storage Tank: 1) exclusive (X), which allows a single client to cache both data and metadata, which it can read and modify; 2) Shared Read (SR), which allows clients to cache data and metadata for read operations, and 3) Shared Write (SW), in which mode clients cannot cache data but can cache metadata in read-only mode. The Storage Tank Protocol also provides copy-on-write capability to support file system snapshots. The server can mark blocks as read-only to enforce copy-on-write.

Storage Tank technology is available today in the IBM Total Storage SAN File System (SANFS). However, this paper and research is based on underlying Storage Tank technology and may not become part of the IBM TotalStorage SAN File System product.

3. Related Work

Data duplication is ubiquitous. Different techniques have been proposed to identify commonality in data, and to exploit this knowledge for reducing storage and network resource consumption due to storing and transferring duplicate data.

Our work was directly inspired by Venti [23]. Venti is a network storage system intended for archival data. In Venti, the unique SHA-1 [12] hash of a block acts as the block identifier, which is used in place of the block address for read and write operations. Venti also implements a write-once policy that prohibits data from being deleted once it is stored. This write-once policy becomes practical, in part, because Venti stores only one copy of each unique data block.

In on-line file systems, performance is essential and data is dynamic and ready to change. This is radically different from the requirements in archival and backup systems, where data is immutable and performance is less of a concern. In our design, duplicate data is also detected on the block level, but in the background. Data is still addressed as usual by the block where it is stored, so data accesses have no extra hash-to-block index searching overheads as in Venti. In turn, it determines that data duplication detection and coalescing is an after-effect effort, *i.e.* it is done after clients have written data blocks to storage devices. The server also maintains a mapping function between block hashes and blocks. A weaker variant of the write-once policy, copy-on-write (COW), is used to guarantee its consistency. Unreferenced blocks due to deletion and COW can be reclaimed.

Single instance store (SIS) [3] also detects duplicate data in an after-effect fashion but on the file level. The technique is optimized for Microsoft Windows remote install servers [18] that store different installation images. In this scenario, the knowledge of file duplication is a priori and files are less likely to be modified. In general on-line file systems, the granularity of file-level data duplication detection may be too coarse because any modification to a file can cause the loss of the benefit of storage reduction.

LBFS [20] and Pastiche [9] detect data duplication at the granularity of variable-sized chunks, whose boundary regions, called *anchors*, are identified by using the techniques of shingling [16] and Rabin fingerprints [24]. This technique is suitable for backup systems and low-bandwidth network environments, where the reduction of storage and network transmission is more important than performance.

Delta compression is another technique that can effectively reduce duplicate data, thus the requirements of storage and network bandwidth [1, 6, 19, 25]. When a base version of a data object exists subsequent versions can be represented by changes (deltas) to save both storage and network transmission. Delta compression, in general, requires some prior knowledge of data object versioning. It cannot explore common data across multiple files and a change of a (base) file may cause recalculating deltas for other files. In DERD (Delta-Encoding via Resemblance Detection) [11], similar files can be identified as pairs by using data similarity detection techniques [4, 16] without having any specific prior knowledge.

Some file systems provide on-line compression capability [15, 21]. Although it can effectively improve storage efficiency, this technique has significant run-time compression and decompression overheads. On-line compression explores intra-file compressibility and cannot take advantage of common data across files.

The techniques of naming and indexing data objects based on their content hashes are also found in several other systems. In the SFS read-only file system [13], blocks are identified by their SHA-1 hashes and the block hashes are hashed recursively to build up more complex structures. The Stanford digital library repository [10] uses the cyclic redundancy check (CRC) values of data objects as their unique handles. Content-derived names [2, 14] take a similar approach to address the issue of naming and managing reusable software components.

4. Design of Duplicate Data Elimination

Our design goal is to transparently reduce duplicate data in Storage Tank as much as possible without penalizing system performance significantly. Instead of finding data duplication at the first spot, we delay this duplication detection and identify and eliminate duplicate data when server loads are low. In this way, we minimize the performance impact of duplicate data elimination (DDE). In our design, we use three techniques: content-based hashing, copy-on-write (COW), and lazy update.

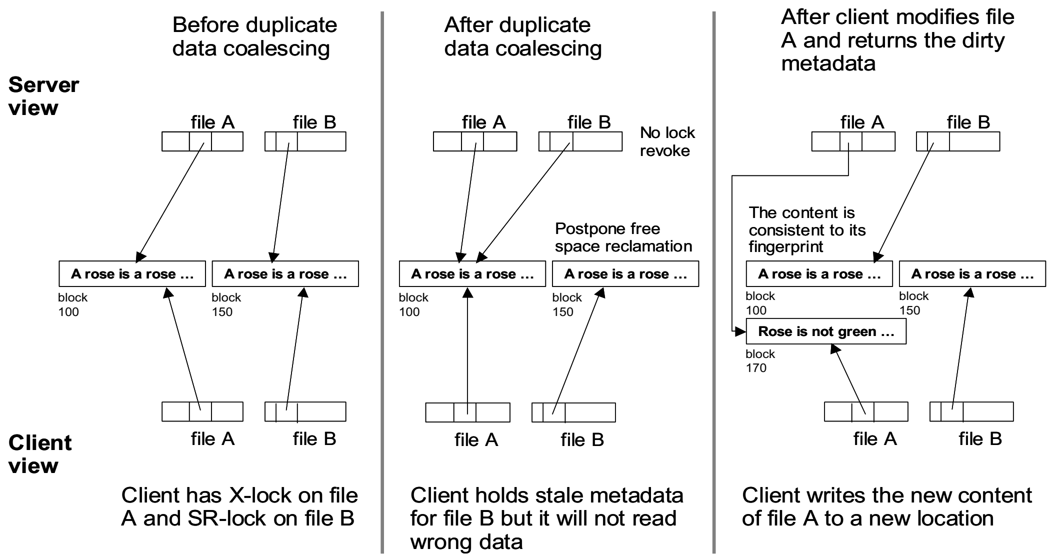
Duplicate data blocks are detected by the Storage Tank server. A client uses a collision-resistant hash function to digest the block contents it writes to storage devices and returns their hashes to the server. Such a unique hash is called the *fingerprint* of a block. The server compares block fingerprints and coalesces blocks with the same fingerprint (and the same content) by changing corresponding file block allocation maps. The server guarantees consistency between block contents and their fingerprints by directing clients to perform copy-on-write. The server also maintains a reference count for each block and postpones the reclamation of unreferenced blocks. These techniques allow the server to update file block allocation maps without revoking any outstanding data locks on them.

Figure 2 shows the basic idea of duplicate data block elimination in a live file system. The client holds an exclusive (X) lock on file A and a shared-read (SR) lock on file B. These lock modes are described in Section 2. File A and B have the same data stored in blocks 100 and 150 respectively. Before duplicate block coalescing, the server and the client share the same view of files and file block allocation maps. The server finds duplicate data and changes the block allocation map of file B to reference block 100 without updating the client. Even though the client has a stale view on file B, it can still read out the correct data because block 150 is not reclaimed immediately. When the client modifies block 100 of file A, it writes the new content to another block and keeps the content of block 100 intact. Therefore, the content and the fingerprint of block 100 are still consistent and file B still references the right block.

4.1. Duplicate Data Detection

The most straightforward and trusted way of duplicate data detection is bytewise comparison. Unfortunately, it is expensive. An alternative method is to digest data contents by hashing them to much shorter fingerprints and detect duplicate data by comparing their fingerprints. As long as the probability of hash collisions is vanishingly small, we can be confident that two sets of data content are identical if their fingerprints are the same.

In our design, duplicate data is detected on the block level, although maintaining a fingerprint for each block imposes a large amount of bookkeeping information on the system. Storage Tank is based on block-level storage devices, and blocks are the basic operation units. Block-level detection avoids unnecessary I/Os required by other approaches based on files [3] or variable-sized chunks [9, 20], in which the client may have to read other disk blocks before it can recalculate fingerprints due to data boundary mis-alignments. Data duplication detection that is based on blocks has finer granularity and, therefore, higher possibility of storage reduction than techniques based on whole files [3]. Approaches based on variable-sized chunks [9, 20] require at least as much bookkeeping information as the block-based approaches because they tend to limit the average chunk size to obtain a reasonable chance of detecting duplicate data. Chunk-based approaches also need a totally different file block allocation map format from the existing



Before duplicate data coalescing
After duplicate data coalescing
After client modifies file A and returns the dirty metadata
Server view
file A
file B
No lock revoke
Postpone free space reclamation
The content is consistent to its fingerprint
A rose is a rose ...
block 100
block 150
Rose is not green ...
block 170
Client view
Client has X-lock on file A and SR-lock on file B
Client holds stale metadata for file B but it will not read wrong data
Client writes the new content of file A to a new location

Figure 2. An example of coalescing duplicate data blocks in a live file system.

Storage Tank implementation, which makes them difficult to employ.

DDE uses the SHA-1 hash function [12] to fingerprint block data contents. SHA-1 is a one-way secure hash function with a 160-bit output. Even in a large system that contains an exabyte of data (10^{18} bytes) as 4 kilobyte blocks (roughly 3×10^{14} blocks), the probability of hash collisions using the SHA-1 hash function is less than 10^{-19}, which is at least 5–6 orders of magnitude lower than the probability of an undetectable disk error. To date, there are no known collisions by this hash function. Therefore we can be confident that two blocks are identical if their SHA-1 hashes are the same. In addition, the system could perform bytewise comparisons before coalescing blocks as a cross check.

In Storage Tank, data and meta-data management are separated, and Storage Tank servers are not involved in the data path during normal operations. Disks have little intelligence and cannot detect duplicate data by themselves. Even with smarter disks, without extensive inter-disk communications, each disk would know only its local data fingerprints, which would reduce the chances of detecting duplication. In our design, a client calculates fingerprints of the blocks it writes to storage devices and returns them to the server. Software implementations of SHA-1 are quite efficient and hashing is not a performance bottleneck. Storage Tank servers have a global view of the whole system and are appropriate for data duplication detection.

4.2. Consistency of Data Content and Fingerprints

After we hash the data content of a block, the fingerprint becomes an attribute of the block. Because the fingerprint is eventually stored on the server and the block can be directly accessed and modified by clients, the consistency of the fingerprint and the data content of a block becomes a problem. In Storage Tank, a client can modify a block by two approaches: update-in-place and copy-on-write.

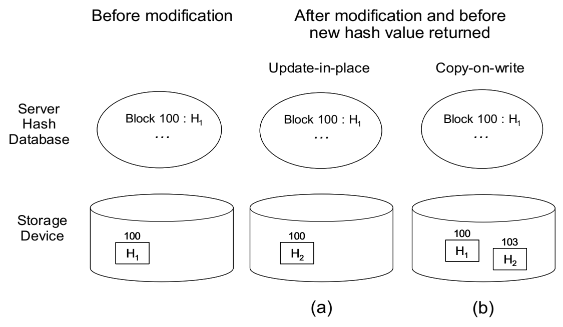


Figure 3. Maintaining consistency between fingerprint and block content under (a) update-in-place and (b) copy-on-write.

4.2.1. Update-in-Place

In Storage Tank, a client can directly modify a block if the block is writable, *i.e.* it writes new data to the same block. This results in inconsistency between the server-side block fingerprint and the block content until the client returns the new fingerprint to the server, as shown in Figure 3(a). The fingerprint of block 100 that the server keeps is inconsistent with the block content until the client returns the latest fingerprint. During the period of inconsistency, any data duplication detection related to this block gives false results.

We can detect this potential inconsistency by checking data locks on the file to which the block belongs. Because the granularity of file data locks in Storage Tank is per file, the server cannot trust all of the block fingerprints of a file with an exclusive data lock on it.

To avoid erroneous duplication detection and coalescing, we could simply delay DDE on those files with exclusive or shared-write locks. This is feasible in some workloads where only a small fraction of files are active concurrently. However, this approach cannot save any storage in some important environments, such as databases, where applications always hold locks on their files.

Another approach is to revoke data locks on files to force clients to return the latest block fingerprints. This causes two technical problems: lock checking and lock revocation. To check file locks, every block has to maintain a reverse pointer to the file to which it belongs, which makes the bookkeeping information of a block even larger. To guarantee consistency between fingerprints and block contents, every block-coalescing operation has to revoke file locks, if necessary, which can severely penalize the system performance. Therefore, eliminating duplicate data blocks under the update-in-place scenario is inefficient, at best, or impossible.

4.2.2. Copy-on-Write

The basic idea of our work is to eliminate duplicate data blocks by comparing their fingerprints. By using a collision-resistant hash function with a sufficiently large output, such as SHA-1, the fingerprints are considered to be distinct for different data. Therefore, the fingerprint can serve as a unique virtual address for the data content of a block. The mapping function from the virtual address to the physical block address is implicitly provided by the block address itself. Our aim is to make the mapping function nearly one-to-one, *i.e.* each virtual address is mapped to only one physical address.

However, update-in-place violates the basic concept of content-addressed storage by making the mapping function inconsistent. Conceptually, if the content of a block is changed, the new content should be mapped to a new block instead of the original one. Consequently, a client should write modified data to new blocks, which implies a write-once policy, as in Venti [23]. However, write-once keeps all histories of data, which is unnecessary and expensive in on-line file systems. Therefore, we use a weaker variant of write-once, copy-on-write. This technique can guarantee the consistency of the mapping function as long as the original blocks are not reclaimed, as shown in Figure 3(b). The fingerprint of block 100 that the server keeps is still consistent with the block content until block 100 is reclaimed.

Apparently, copy-on-write has a noticeable overhead on normal write operations. Every block modification requires free block allocation because the modified content needs to be written to a new block. However, this cost is less significant than we thought. Some applications, such as Emacs and Microsoft Word, write the whole modified file into a new place, in which case there is no extra cost for COW. The server could also preallocate new blocks to the client that acquires an exclusive or shared-write lock. The most promising approach to alleviate the extra allocation overhead of COW is for the clients to maintain a private storage pool on behalf of the server from which they could allocate locally. Therefore, there is almost no extra cost for COW.

4.3. Lazy Lock Revocation and Free Space Reclamation

The server coalesces duplicate data blocks and reduces storage consumption by updating file block allocation maps to point to one copy of the data and reclaim the rest. During file block allocation map updates, the server does not check whether any client holds data locks on the files. Therefore, the block allocation maps held by the server and clients can be inconsistent, as illustrated in Figure 2. However, we postpone the reclamation of the dereferenced blocks. Therefore, clients holding stale file block allocation maps can still read the correct data from these blocks. At some particular time, *e.g.* midnight, or when the file system is running low on free space, the server revokes all data locks held by clients and frees those dereferenced blocks.

5. Process of Duplicate Data Elimination

Duplicate data elimination is done by the coordination between clients and servers. Simply speaking, clients perform copy-on-write operations and calculate and return block SHA-1 hashes to servers. Servers log clients' activities and identify and coalesce duplicate data blocks in the background. Users are unaware of such operations.

5.1. Impact on Client's Behaviors

In addition to its normal behaviors, a client calculates SHA-1 fingerprints for the data blocks it writes and returns the fingerprints to the server. Because copy-on-write is used (Section 4.2.2), the client does not write modified data blocks back to their original disk blocks; instead, modified data is written to newly-allocated blocks. As long as the client holds a file data lock, further modifications to the same logical block are written to the same newly-allocated disk block. On an update to the server, the client sends the latest block fingerprints along with the block logical offsets within the file and the original physical locations of modified data blocks.

5.2. Data Structures on the Server

We discuss necessary data structure supports on the server that facilitate duplicate data block detection and elimination. Essentially, a reference count table, a fingerprint

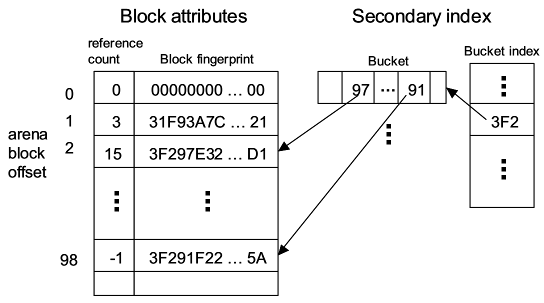


Figure 4. Data structures for storing and retrieving block attributes.

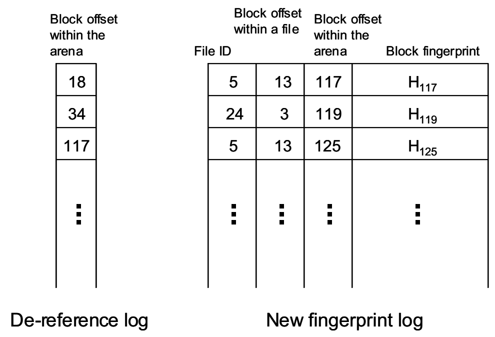


Figure 5. Data structures for logging recent clients' activities.

table and its secondary index maintain attributes associated with blocks, *i.e.* reference counts and fingerprints, as shown in Figure 4; and a dereference log and a new fingerprint log record recent clients' activities, as shown in Figure 5.

The scope of duplicate data elimination is an important design decision. The larger the scope, the higher the degree of data duplication can be, thus providing more benefit. However, for various reasons, people may want to share data only within their working group or their department. Therefore, we limit data duplication detection and elimination within a file set, which essentially is a logical subset of the global namespace. Even within a file set, files can be stored in different storage pools that may belong to different storage classes, which have different characteristics in access latency, reliability, and availability. Sharing data across storage classes can result in noticeable impacts on the quality of storage service. Therefore, we further narrow the scope of DDE within an arena, which provides the mapping between a file set and a storage pool. The data structures we will discuss soon are per arena.

Data are not equally important. Detecting and coalescing temporary, derived, or cached data is less beneficial. Because Storage Tank provides policy-based storage management, a system can be easily configured to store these data in less reliable and so cheaper storage pools, while storing important data in more reliable storage pools. DDE within an arena can take advantage of this flexibility.

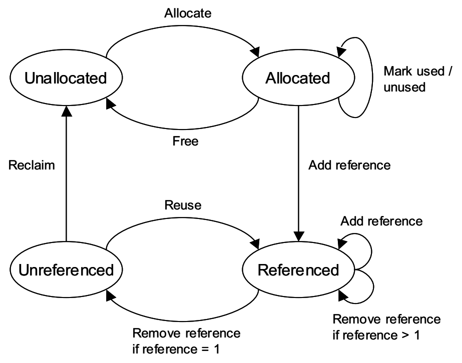


Figure 6. A block is in one of the following states: unallocated, allocated, referenced, and unreferenced.

Because an arena allows the logical to physical mapping of space in the file set to the LUNs in the storage pool, it is equivalent to referencing a block by its physical location and by its logical offset within an arena. For convenience, we will reference an allocated physical block by its logical offset within the arena.

To keep the per-arena data structures to an optimal size, we use 32-bit integers to represent logical block offsets within an arena. Therefore, an arena can contain no more than 2^{32} physical blocks. For 4 KB blocks, an arena can manage 16 TB storage, which is large enough for most applications and environments. However, there is no such a limitation on the capacity of a file set because it can cross multiple storage pools and can consist of multiple arenas.

5.2.1. Reference Count Table

With block coalescing and sharing, a physical block can be referenced multiple times by different files or even one file. Therefore, a reference count is necessary for each block in the arena. From the viewpoint of DDE, a block can be in one of the following four states: 1) free – the block is unallocated; 2) allocated – the block is allocated but unused or it contains valid data that is unhashed; 3) referenced – the block contains valid data, which has been hashed, and is referenced at least once; and 4) unreferenced – the block is allocated and hashed, but has no file referencing it and can be freed or reused. Figure 6 illustrates the four states and the transitions among them.

Without accessing the arena block allocation map, we cannot know whether a block is allocated or not. Fortunately, we are interested in the validity of block fingerprints in our work. Therefore, blocks that are in the first two states (free and allocated) can be merged into one state, invalid, because they contain no valid fingerprints. The reference count of a block indicates its state: 1) invalid, where the reference count is 0, 2) referenced, where the reference count is no less than 1, or 3) unreferenced, where the reference count is -1. The initial state of a block is invalid because it contains no valid fingerprint until a client calculates it.

A reference count table keeps the reference count for each block in the arena. The table is organized as a linear array, which is indexed by the 32-bit logical block offset within the arena. Each entry in this table is also a 32-bit integer, indicating the state of the corresponding block. The size of the table is up to $2^{32} \times 4$ bytes $= 16$ GB. Because block reference counts are crucial to data integrity, any update on them should be transactional.

A block may contain valid data but has no fingerprint associated with it. Note that the fingerprint of a block is calculated when it is written. If a block is written before the server turns on this feature, it has no fingerprint on the server. A utility running on the server could ask clients to read those data blocks and calculate their fingerprints on behalf of the server.

5.2.2. Fingerprint Table

The fingerprint table keeps unique fingerprints of the blocks in an arena. Each fingerprint in this table is associated with a unique physical block. In other words, the table maintains a one-to-one mapping function between fingerprints and physical blocks. We detect and coalesce duplicate data blocks when we merge the new fingerprint log to this table. The fingerprint table is also organized as a linear array and indexed by the 32-bit logical block offset. Each entry contains a 160-bit SHA-1 fingerprint. A fingerprint is valid only if its block reference count is no less than 1.

The size of the table is up to $2^{32} \times 20$ bytes $= 80$ GB, and it cannot fit in memory. Fortunately, disk block accesses have sequential patterns due to sequential block allocations and file accesses. Therefore, we organize the secure fingerprint table linearly to facilitate comparisons under sequential block accesses. If two disk blocks contain the same content, it is likely that their consecutive blocks also have identical contents. The linear structure makes consecutive fingerprint comparisons efficient because all related entries are in memory.

Conceptually, both the reference count table and the fingerprint table are for describing block attributes and can be merged into one table. Because their sizes are potentially large, and the block reference count is accessed more frequently than the fingerprint, we store them separately to optimize system memory usage.

5.2.3. Secondary Index to Fingerprint Table

Although the linear fingerprint table favors sequential searching, it is difficult to look for a particular fingerprint in this table. Therefore, we also index the table by partial bits of the SHA-1 fingerprint to facilitate random searching. A static hash index is used for this purpose. The hash buckets are indexed by the first 24 bits of the SHA-1 fingerprint. Each bucket contains a 32-bit block pointer. Therefore, the size of the first level index is $2^{24} \times 4$ bytes $= 64$ MB, which can well fit in memory. Each entry in the bucket block contains a 32-bit in-arena logical block offset, indicating the block that the fingerprint is associated with, and the next 32 bits of the SHA-1 fingerprint. Because an arena contains no more than 2^{32} blocks, the average number of hash entries in a bucket is $2^{32}/2^{24} = 2^8 = 256$. When the bucket block size is 4 KB, the average block utilization is $(256 \times (4+4))/4096 = 50\%$. For an arena with capacity much less than 16 TB, multiple buckets can share one bucket block for better storage and memory utilization.

5.2.4. Dereference Log

The dereference log records the in-arena logical offsets of blocks that are recently deleted, dereferenced due to COW by clients, or dereferenced due to block coalescing by the server. We will discuss the third case in greater details in Section 5.3.3. This log is also called *semi-free list* because the blocks in this list could be freed if there is no longer any reference to them. Each entry in this log is a 32-bit integer. To avoid storage leakage, any update on this log should be transactional.

5.2.5. New Fingerprint Log

The new fingerprint log records clients' recent write activities. Each entry in this log includes a 64-bit file ID, a 64-bit logical block offset within the file, a 32-bit logical offset within the arena, and a 160-bit SHA-1 fingerprint. Appending new entries to this log can be non-transactional for the performance purpose because losing the most recent entries only causes losing some opportunities for storage reduction.

5.3. The Responsibilities of the Server

The server enforces a client's copy-on-write behaviors by marking the copy of the file block allocation map in the message buffer as read-only when it responds to a file data lock request from the client. The server immediately logs clients' recent activities, such as delete and write operations. In our design, DDE runs in a best-effort fashion. Therefore, the server lazily detects and coalesces duplicate data blocks and reclaims unused blocks. It also maintains block reference counts and fingerprints correspondingly.

5.3.1. Logging Recent Activities

When the server receives a dirty file block allocation map from a client, it compares it with the one on the server. If a block is marked as unused due to copy-on-write, the server first checks whether it is still referenced by the server-side file block allocation maps. If referenced, the in-arena logical offset of the unused block is appended to the dereference

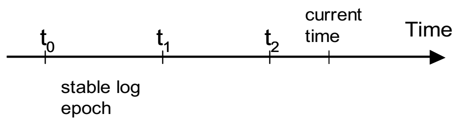


Figure 7. Log epochs.

log; if not (it is possible due to duplicate block coalescing without lock revocation), the block currently referenced by the server-side block allocation map is logged because it is dereferenced by recent modifications on the corresponding file logical block. The unused block returned from the client is not logged because it has been dereferenced due to block coalescing. Other unused blocks are logged.

For each recently-written block, the server also appends an entry to the new fingerprint log, including the identifier of the file to which it belongs, its logical block offset within the file, its logical block offset within the arena, and its fingerprint.

5.3.2. Log Epoch and Preprocessing

We periodically checkpoint the dereference log and the new fingerprint log for two reasons. First of all, the data duplication detection and elimination processes run in a best-effort and background fashion and are unlikely to keep up with the most recent clients' activities. Logging and checkpointing these activities allow the server to detect and coalesce duplicate data blocks during its idle time. By checkpointing the log to epochs, we also limit the number of activities the server processes at a time. Second, recently-written blocks are likely to be modified again. Trying to coalesce these blocks is less beneficial. Therefore, we try to coalesce only blocks in a stable epoch, as shown in Figure 7, whose lifetimes have been long enough.

Assume that we want to merge the new fingerprint log in epoch (t_0, t_1) to the fingerprint table. Because random accesses on the fingerprint table are expensive, we try to reduce the number of fingerprint comparisons by deleting unuseful entries in the new fingerprint log in epoch (t_0, t_1).

First of all, we find overwritten file logical blocks by sorting the log by file ID and logical block offset within a file. We delete the older entries from the log and set their block reference counts to be -1 (unreferenced). We set the block reference counts of other entries in the log to be 1. Second, we scan the dereference log in epoch (t_0, t_1). For each entry in the log, we decrease its block reference count by 1; if the count becomes less than 1, we set it to be -1. Third, we compact the new fingerprint log (t_0, t_1) by deleting those entries also in the dereference logs (t_0, t_1) and (t_1, t_2). The matched entries in the dereference log (t_1, t_2) are removed and their block reference counts are set to be -1.

Note that a fingerprint in the fingerprint table becomes invalid when its block reference count reaches -1. Conceptually, we should also remove its index entry in the secondary index. However, we do not update the secondary index during the log preprocessing because of performance reasons. We postpone the removal of false indexes until the duplicate block coalescing process notices them. False index removal also happens when a secondary index bucket block becomes full.

5.3.3. Merging to Fingerprint Table

We detect and coalesce duplicate data blocks when we merge the compacted new fingerprint log to the fingerprint table. Figure 8 shows the processes of duplication detection and coalescing.

For each entry in the log, we first check whether it has a matching fingerprint in the fingerprint table. If not, we insert the new fingerprint into the table and update the secondary index. If there is an identical fingerprint in the fingerprint table, we check the validity of the fingerprint. If the block reference count of the fingerprint is less than 1, the primary block in the fingerprint table contains no valid data. Therefore, we insert the new fingerprint into the table and update the secondary index. We also need to delete the false index in the secondary index because the previous matching index leads to a block containing invalid data. If the block reference count is no less than 1, we fetch the block allocation map of the file to which the recently-written block belongs, and check whether this block is still referenced by this file. If not, this means that the corresponding logical block in this file was modified after the current fingerprint was returned, and we simply discard this coalescing operation. If the block is still referenced, we update the file block allocation map by referencing the primary block in the fingerprint table without checking or revoking data locks on this file. We also increase the reference count of the primary block and set the reference count of the coalesced block to be -1. When a block is inserted into the fingerprint table, either by adding a new entry or by coalescing to another block, it is marked read-only in the block allocation map of the file to which it belongs.

5.3.4. Free Space Reclamation

A free space reclamation process scans the reference count table in the background. It logs the addresses of the blocks with reference counts -1 and sets their reference counts to 0. At some particular time, *e.g.* midnight, or when the file system is running low on free space, it revokes all data locks and free these blocks.

6. Case Studies

We examined six data sets and studied their degrees of data duplication and their compressibility under common compression techniques. The data sets are summarized in Table 1.

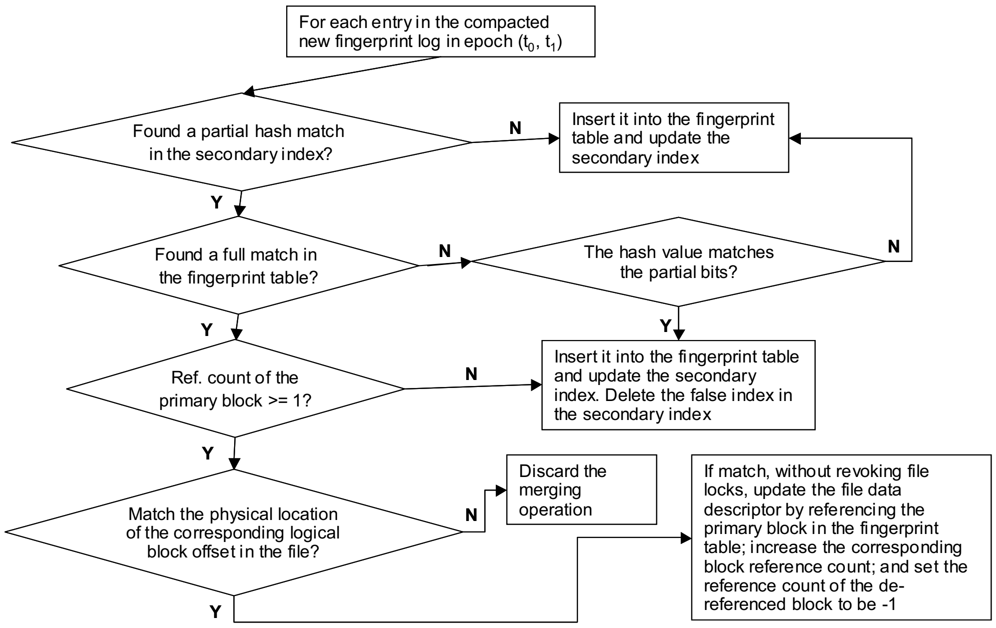

For each entry in the compacted new fingerprint log in epoch (t_0, t_1)
Found a partial hash match in the secondary index?
N
Insert it into the fingerprint table and update the secondary index
Y
Found a full match in the fingerprint table?
N
The hash value matches the partial bits?
N
Y
Y
Ref. count of the primary block >= 1?
N
Insert it into the fingerprint table and update the secondary index. Delete the false index in the secondary index
Y
Discard the merging operation
If match, without revoking file locks, update the file data descriptor by referencing the primary block in the fingerprint table; increase the corresponding block reference count; and set the reference count of the de-referenced block to be -1
Match the physical location of the corresponding logical block offset in the file?
N
Y

Figure 8. Merge the new fingerprint log to the fingerprint table.

Table 1. Data sets

Name	Description	Size (GB)	Number of files
SNJGSA	file server used by a development team	57	661,729
BVRGSA_BUILD	file server used by the development team for code build	344	2,393,795
BVRGSA_TEST	file server used by the development team for testing	215	115,141
GENOME	human being genome data	348	889,884
LTC_MIRROR	local mirror of installation CDs for different Linux versions	261	241,724
PERSONAL_WORKSTATIONS	aggregation of ten personal workstations	123	879,657

The first three data sets—SNJGSA, BVRGSA_BUILD, and BVRGSA_TEST—are from file servers used by the Storage Tank development team. The servers are used to distribute and exchange data files, but are not used to hold or archive the primary copy of important files. The first server, SNJGSA, is a remote replica that holds a subset of the daily builds that are stored on BVRGSA_BUILD, and a subset of the test data that is on BVRGSA_TEST. In general, the oldest files are deleted when the servers run low on space. The files are almost never overwritten; they tend to be created and then not modified until they are purged few months later.

GENOME contains the human being genome sequence, and is used at UCSC by various bioinformatics applications. The genomic data is encoded in letters, when some single letters and some letter combinations can repeat thousands to millions of times but in fine granularities.

LTC_MIRROR is a local ftp mirror of the IBM Linux Technology Center (LTC) that includes Open Source and IBM software for internal download and use. Among other things, the ftp site holds the CD images (iso) of different Red Hat Linux installations starting at RH7.1 up to RH9.

The last data set is an aggregation of ten personal workstations at the IBM Almaden Research Center that are running Windows. All systems are used for development as well as for general purposes such as email, working documents, etc. We also present the results of these systems when they are analyzed separately.

For each data set, we collected the size and the number of files of the system. We calculated the amount of storage that is required after eliminating data duplication at the granularity of 1 KB blocks. To compare DDE with common compression techniques, we collected the compressed file sizes, using LZO on 64 KB data blocks. LZO (Lempel-Ziv-Oberhumer) is a data compression library that favors speed over compression ratio [22]. We empirically found that the compression capability of LZO is similar to other techniques' when the block size is large. In addition, we calculated the storage reduction achieved by combining the techniques of DDE and LZO. We also calculated what per-

Table 2. DDE and compression results.

Name	% of storage required after DDE (1 KB blocks)	% of storage required after eliminating whole file duplications	% of storage required after LZO on 64 KB blocks	% of storage required after combining DDE and LZO on 64 KB blocks
SNJGSA	32%	55%	56%	30%
BVRGSA_BUILD	21%	62%	67%	35%
BVRGSA_TEST	69%	85%	53%	47%
GENOME	96%	98%	46%	44%
LTC_MIRROR	80%	94%	98%	89%
PERSONAL_WORKSTATIONS	54%	69%	63%	43%

Table 3. DDE and compression results for personal workstations.

System	% of storage required after DDE (1 KB blocks)	% of storage required after eliminating whole file duplications	% of storage required after LZO on 64 KB blocks	% of storage required after combining DDE and LZO on 64 KB blocks
1	66%	71%	61%	43%
2	61%	78%	57%	43%
3	63%	77%	55%	41%
4	77%	91%	63%	55%
5	67%	92%	62%	53%
6	69%	77%	58%	48%
7	70%	80%	61%	49%
8	71%	78%	71%	55%
9	87%	91%	80%	74%
10	73%	84%	57%	48%

centage of the storage is still required after eliminating only whole file duplications.

Table 2 shows the percentage of storage required for different data sets after using DDE at the granularities of 1 KB blocks and whole files, LZO on 64 KB blocks, and the combination of DDE and LZO. DDE at 1 KB blocks only requires one-fifth to one-third of the original storage to hold the BVRGSA_BUILD and SNJGSA data sets and it achieves one to two times higher storage efficiency (the ratio of the amount of logical data to the amount of required physical storage) than LZO at 64 KB blocks. This is because both data sets contain daily builds of the Storage Tank codes that share lots of codes among versions; also the data sets include very small files on average, making LZO inefficient. BVRGSA_TEST on the other hand, has on average larger files (1.9 MB) and the best reduction is achieved when using the combination of LZO compression and DDE on 64 KB blocks. We will study the data set of SNJGSA in greater detail in Section 6.1.

The genomic data set is encoded in letters. Some single letters and letter combinations can repeat thousands to millions of times but at quite fine granularities. With this type of data set it is unlikely to find duplications at the granularities of kilobytes and DDE cannot improve storage efficiency significantly. On the other hand, common compression techniques, such as LZO, are suitable for this data set. The 4% of reduction by DDE is partially due to common file headers and a common "special letter pattern" that fills in gaps in the genomic sequence and partially due to duplicated files that were created by analyzing applications.

Using LZO on the data set of LTC_MIRROR barely reduces storage consumption because generally the files in the Linux installation CD images have already been packaged and compressed. DDE can take advantage of cross-file duplications, mainly from different installation versions, and reduce storage consumption by 20%. A more interesting data set is the actual installation of different Linux versions, instead of the installation images. We plan to look into that in the nearest future.

We also studied the storage requirements of ten personal workstations after using different techniques on individual systems, as shown in Table 3. On average 70% of storage is required for individual systems after applying DDE on them, from which more than half of the saving is due to eliminating whole file duplications. LZO compression can provide better storage efficiency on these systems. By combining both LZO and DDE on 64 KB blocks, the percentage of storage required for individual systems is further reduced to 51% on average. When aggregating files from all machines together, which is potentially what happens with enterprise-level backup applications, the aggregated storage requirement of ten personal workstations by using DDE drops significantly, from 70% to 54%, as shown as PERSONAL_WORKSTATIONS in Table 2. This is because the more individual systems are aggregated, the higher the degree of data duplication is likely to be. Consequently, DDE can result in tremendous storage savings for back up systems that potentially backup hundreds to thousands of personal workstations on a daily basis, which shows a very good example of an application that could benefit from data duplication elimination at the file system. Combining DDE and LZO on 64 KB blocks can further reduce the storage requirement to 43%.

The granularity of duplicate data detection affects the effectiveness of DDE. Table 2 shows that file-level detection can lose up to 50–70% of the chance of finding duplicate data at 1 KB blocks. We also noticed that using both DDE and LZO on 64 KB blocks does not always generate better results than using DDE solely. In general, smaller block sizes favor DDE because of finer granularities on duplication elimination; larger block sizes favor LZO because of more efficient encoding. In fact, DDE and LZO-like compression techniques are orthogonal when they are operated at the same data unit granularity because compression explores intra-unit compressibility and DDE explores inter-unit duplications.

6.1. Detailed Study on SNJGSA

Figure 9 shows the results of applying DDE to the SNJGSA data set using a range of file system block sizes. The results are given as a percentage of the data blocks that are unique. The block size varies from 512 bytes to 64 KB. As expected, the smallest block size works best because it enables the detection of shorter segments of duplicate data. Interestingly, DDE's effectiveness starts to improve again when the block size reaches 32 KB. The reason is that the file system is "wasting" space using these larger blocks, and DDE is proportionally coalescing more of this wasted space. This effect begins to outpace the reduced number of duplicate blocks due to coarser block granularities. Note that Storage Tank does not support blocks sizes smaller than 4 KB; these results are included to show the savings we are missing out on. The additional space saving potential of smaller blocks is modest: 5%, for instance, between 1 KB and 4 KB blocks in the SNJGSA1 data set.

SNJGSA's usage pattern and its "FIFO" style space management allow us to use this data to simulate a growing file system. Figure 10 shows the amount of space saved by DDE as the file system, which starts empty, grows to eventually contain all the files in the SNJGSA data set. The files are added in order of their modified times (mtime). 4 KB blocks are used. The rightmost value on the chart is 38%, which matches the space savings shown in Figure 9.

If Figure 10 had shown a gradual improvement starting at 100% (no compression) and monotonically decreasing to 38%, we could assert that duplication in this data set is independent of time. This is the type of curve that would be generated if the files were inserted in random order rather than being sorted by mtime. If the chart had shown a flat curve, this would indicate that data duplication is highly correlated in time, *i.e.* has strong temporal locality. The duplicate blocks found in a new file would likely match blocks that were added, for instance, within the last 24 hours, but very unlikely to match blocks that were added a month ago. DDE performs most efficiently on highly time correlated duplication because duplicate blocks tend to be coalesced within a log epoch and require fewer updates to the fingerprint table. At the same time, uncorrelated duplication allows DDE to produce a continually improving compression ratio as the data set grows.

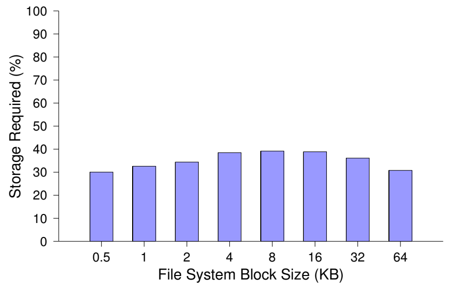


Figure 9. SNJGSA, percentage of storage required after removing duplicate data.

The SNJGSA data set consists of 15 million 4 KB blocks, among which 3.3 million are referenced only once. Among the remaining 11.7 million blocks, only 2.5 million are unique. Figure 11 shows the cumulative distributions of storage savings made by frequently occurring data blocks. It reveals that only 1% of unique blocks contributes to 29% of the total storage savings. This suggests us that even a small amount of hash cache on the client side could explore the frequency of data duplication and save actual I/O on the first spot. Although we have not had the opportunity to study the recency of data duplication, we believe that with careful design, the client-side hash cache can also take advantage of this recency to improve system performance. Furthermore, the spatial and temporal localities of duplicate data generation are dominant factor of DDE's performance on the server side.

SNJGSA and BVRGSA_BUILD demonstrate applications that can substantially benefit from DDE. The engineers using these file servers exploit storage space to make their jobs easier. One way they do this is by creating hundreds of "views" of their data in isolated directory trees, each with a dedicated purpose. However, these pragmatic, technology savvy users do not use a snap-shot facility to create these views, or a configuration management package, or even symbolic links. The engineers employ the most robust and familiar mechanisms available to them: copying the data, applying small alterations, and leave the results to linger indefinitely. In this environment, the file system is the data management application, and DDE extends its reach.

7. Discussions

Often duplicate data copies are made for the purpose of reliability, preservation, and performance. Typically, such duplicate copies are and should be stored in different stor-

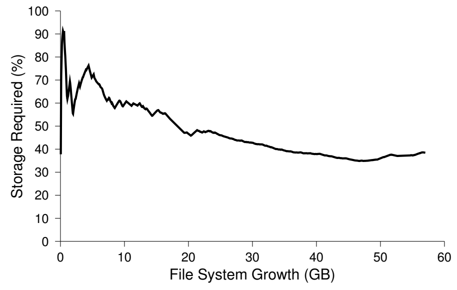


Figure 10. SNJGSA, percentage of unique blocks in a simulated growing file system.

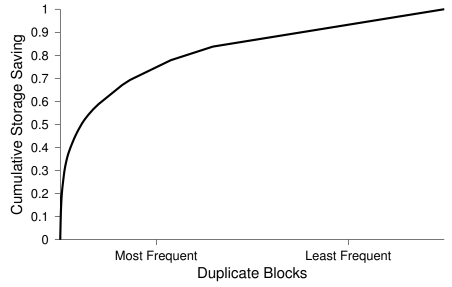


Figure 11. SNJGSA, cumulative contribution to the total storage savings made by coalescing frequently occurring data blocks. Blocks are sorted by their reference frequencies.

age pools. Since the technique of duplicate data elimination is scoped to an arena, data on different storage pools will not be coalesced and its reliability, preservation, and the performance of accessing data will not be affected. If the duplicate data copies are made against inadvertent deletion or corruption of file data, DDE allows copies to be made without consuming additional storage space and still protects against deletion or corruption. However, DDE does create an exposure to potential multiple file corruptions due to a single block error.

DDE coalesces duplicate data blocks in files. It can result in file fragmentation thus degrade the system performance due to non-contiguous reads. This can be alleviated by only coalescing duplicate data with sizes of at least *N* contiguous blocks. In Section 6, we also found that the majority of duplicate blocks come from whole files, in which cases reading those files has no additional seek overheads. It is probable that DDE can reduce system write activities if clients can detect duplicate data before writing to storage devices, which will be discussed further in Section 8. DDE can also potentially improve the storage subsystem cache utilization because only unique data blocks will be cached.

The degree of data duplication can vary dramatically in different systems and application environments. DDE is a technique that is suitable for those environments with expected high degrees of data duplication such as backup of multiple personal workstations, and it is not necessarily intended for general uses.

The key techniques used in DDE are content-based hashing, copy-on-write, and lazy updates. With necessary and appropriate supports, these techniques thus DDE could be also applicable to other file systems besides Storage Tank. Particularly, lazy updates minimize the performance impact of identifying duplicate data and maintaining block metadata, which will also be beneficial to other file systems.

8. Future Work

We are working on implementing duplicate data elimination in Storage Tank. Besides implementation, there are several research directions we can explore in the future.

The technique of copy-on-write plays a key role in our design to guarantee consistency between fingerprints and block contents. However, it forces a client to request new block allocations for each file modification and has noticeable overheads on normal write operations. To alleviate the extra allocation cost due to COW, the server could preallocate new blocks to a client that acquires an exclusive or shared-write lock. Therefore, the preallocation policy for COW is an interesting research topic. Another promising approach to alleviate this overhead is to allow a client to maintain a small private storage pool on behalf of the server. Therefore, there is almost no extra cost for COW.

In our current design, we employ quite a naive coalescing policy: when we find a recently-written block containing the same data as a block in the fingerprint table, we simply dereference the new block and change the corresponding file block pointer to the primary block in the fingerprint table. This policy is suboptimal in terms of efficiency. Although we have considered file and block access patterns in our design, we do not, for simplicity, explicitly elaborate policies that favor sequential fingerprint probing and matching under certain block access patterns. Therefore, further research on such policies is needed.

The naive coalescing policy we describe in this paper may also result in file fragmentation. Coalescing few blocks within a large file is less desirable. Therefore, a study on policies for minimizing file fragmentation is interesting. Furthermore, a good coalescing policy could reduce storage fragmentation by reusing the lingering unused blocks due to lazy free space reclamation.

The fingerprint of a block is a short version of its data. A client can easily keep a history of its recent write activities by maintaining a fingerprint cache. The client can do part of the duplicate data elimination work in conjunction with the server. More beneficially, actual write operations to storage can be saved if there are cache hits.

As far as we know, there are no extensive and intensive studies on the duplicate data distributions of the block level or other levels. A better understanding of data duplication in file systems can be enormously beneficial for making good duplicate data coalescing policies.

In our design, we add two attributes on physical disk blocks: the reference count and the SHA-1 fingerprint of the block content. We also provide appropriate data structures to store and retrieve these attributes. Our work makes it feasible to check data integrity in Storage Tank. A client can ensure that the data it reads is the data it writes by comparing the fingerprint on the server with the one calculated from the data it recently reads.

9. Conclusions

Although disk prices drop dramatically, storage is still a precious resource in computer systems. For some data sets, reducing storage consumption caused by duplicate data can significantly improve storage usage efficiency. By using techniques of content-based hashing, copy-on-write, lazy lock revocation, and lazy free space reclamation, we can detect and coalesce duplicate data blocks in on-line file systems without a significant impact on system performance. Our case studies show that 20–79% of storage can be saved by the technique of duplicate data elimination at 1 KB blocks in some application environments. File-level duplication detection is sensitive to changes of file contents and can lose up to 50–70% of the chance of finding duplicate data at finer granularities.

Acknowledgments

We are grateful to Robert Rees, Wayne Hineman, and David Pease for their discussions and insights in Storage Tank. We thank Scott Brandt, Ethan Miller, Feng Wang, and Lan Xue of the University of California at Santa Cruz for their helpful comments on our work. We thank Vijay Sundaram of the University of Massachusetts at Amherst, in particular, for his invaluable discussions. We also thank Terrence Furey and Patrick Gavin of the bioinformatics department of UCSC for providing access to the GENOME data set and helping us to understand the results. Our shepherd Curtis Anderson provided useful feedback and comments on our first draft.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, 49(3):318–367, May 2002.

[2] K. Akala, E. Miller, and J. Hollingsworth. Using content-derived names for package management in Tcl. In *Proceedings of the 6th Annual Tcl/Tk Conference*, pages 171–179, San Diego, CA, Sept. 1998. USENIX.

[3] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. USENIX, Aug. 2000.

[4] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES '97)*, pages 21–29. IEEE Computer Society, 1998.

[5] R. Burns. *Data Management in a Distributed File System for Storage Area Networks*. Ph.d. dissertation, Department of Computer Science, University of California, Santa Cruz, Mar. 2000.

[6] R. Burns and D. D. E. Long. Efficient distributed back-up with delta compression. In *Proceedings of I/O in Parallel and Distributed Systems (IOPADS '97)*, pages 27–36, San Jose, Nov. 1997. ACM.

[7] R. Burns, R. Rees, and D. D. E. Long. Safe caching in a distributed file system for network attached storage. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*. IEEE, May 2000.

[8] R. Burns, R. Rees, L. J. Stockmeyer, and D. D. E. Long. Scalable session locking for a distributed file system. *Cluster Computing Journal*, 4(4), 2001.

[9] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, Boston, MA, Dec. 2002.

[10] A. Crespo and H. Garcia-Molina. Archival storage for digital libraries. In *Proceedings of the Third ACM International Conference on Digital Libraries (DL '98)*, pages 69–78, Pittsburgh, Pennsylvania, June 1998. ACM.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, June 2003.

[12] Secure hash standard. FIPS 180-1, National Institute of Standards and Technology, Apr. 1995.

[13] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 181–196, San Diego, CA, Oct. 2000.

[14] J. Hollingsworth and E. Miller. Using content-derived names for configuration management. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 104–109, Boston, MA, May 1997. IEEE.

[15] S. Kan. ShaoLin CogoFS – high-performance and reliable stackable compression file system. http://www.shaolinmicro.com/product/cogofs/, Nov. 2002.

[16] U. Manber. Finding similar files in a large file system. Technical Report TR93-33, Department of Computer Science, The University of Arizona, Tucson, Arizona, Oct. 1993.

[17] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank—a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[18] Microsoft Windows 2000 Server online help file. Microsoft Corporation, Feb. 2000.

[19] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of SIGCOMM97*, pages 181–194, 1997.

[20] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Oct. 2001.

[21] R. Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly and Associates, 1997.

[22] M. F. Oberhumer. oberhumer.com: LZO data compression library. http://www.oberhumer.com/opensource/lzo/, July 2002.

[23] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In D. D. E. Long, editor, *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[24] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[25] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*, pages 87–95, Stockholm, Sweden, Aug. 2000. ACM Press.