File System Support for Delta Compression

Joshua P. MacDonald
University of California at Berkeley,
Department of Electrical Engineering and Computer Sciences,
Berkeley, CA 94720, USA
jmacd@cs.berkeley.edu

Abstract

Delta compression, which consists of compactly encoding one file version as the result of changes to another, can improve efficiency in the use of network and disk resources. Delta compression techniques are readily available and can result in compression factors of five to ten on typical data, however managing delta-compressed storage is difficult. I present a system that attempts to isolate the complexity of delta-compressed storage management by separating the task of version labeling from performance issues. I show how the system integrates delta-compressed transport with delta-compressed storage.

Existing tools for managing delta-compressed storage suffer from weak file system support. Lacking transaction support, the traditional file system has only one atomic operation which forces unnecessary disk activity due to copying costs. I demonstrate that file system transaction support can improve application performance and extensibility with the benefit of strong, fine-grained consistency.

1 Introduction

The practice of *delta compression*, compactly encoding a file version as the mutation (*delta*) of an existing version, has important applications in systems today. In a version control system it is common to store related file versions using this technique to save disk space while maintaining a complete modification history. More importantly, however, delta compression can be used to reduce network transmission time. Distribution agents of all kinds can benefit from this technique whenever transporting data that is significantly similar to a previous transmission. Software distribution systems, Web transportation infrastructure, and version control systems can employ delta compression to make more efficient use of network and disk resources, with compression factors of five to ten typical for some data.

The basic mechanics in a delta-compression system can be easily understood. We have algorithms that quickly compute near-optimal deltas, and the decoding process usually is simple. To manage delta-compressed storage, however, takes organization beyond the encoding and decoding of deltas, and the operation of delta-compressed transport can be even more difficult. This involves deciding upon an encoding relationship

between versions, which has time–space implications, as well as a version labeling interface. Furthermore, there are potential benefits to managing delta-compressed storage and transport simultaneously, with the use of pre-computed deltas. I will present the design of a delta-compression system that addresses these issues yet has a simple interface.

Existing tools for managing delta-compressed archives suffer from weak file system support. Lacking transaction support, the traditional file system has only one atomic operation which forces unnecessary disk activity due to copying costs. For a delta-compression system using a single-file archive encoding, the cost of an atomic operation grows with the total archive size, causing insertion time to degrade. My system avoids this problem with the use of transactions, supporting insertion time that is independent of total archive size. In this manner, I will show that file system transaction support can improve application performance because weak consistency semantics force inefficient behavior in some applications.

In addition to improving insertion-time performance, transactions permit extensibility by supporting independent failure recovery among composed modules. My system includes an extensible, application-level file system layer that separates version labeling from the complex administrative tasks of delta-compressed storage. After describing the delta-compressed storage manager, the operations it supports, and the algorithms it uses, I will describe how the file system layer gives it an efficient, modular implementation through the use of transactions.

2 Delta-Compressed Storage

To avoid later confusion, we shall use the following definitions. *Text compression* is a lossless transformation of a single file's contents to require less space or transmission time. *Delta compression* consists of representing a *target* version's contents as the mutation (*delta*) of some existing *source* contents to achieve the same goal, a reduction in space or time. Typically, the target and source are related file versions and have similar contents. The compression, or α, of a delta is its size divided by the size of its target version.

One of the primary applications of delta compression is for efficiently storing multiple versions of a file in an archive. Version control systems typically support this type of compression for maintaining complete file histories. RCS [50] and SCCS [39] are two traditional systems that provide version control with delta-compressed storage for a single file. More advanced version control systems such as CVS [3] and PRCS [29] have emphasized the importance of providing version control over a collection of files, as opposed to a single file, resulting in the use of RCS and SCCS for back-end storage rather than for version control. This discussion assumes that these tools are being used by an application, not directly by a user.

Delta-compressed storage is achieved by storing deltas in place of the versions they represent. A plain-text, *literal* version of the file is stored in an archive along with deltas allowing the other versions to be reconstructed. Deltas are sometimes organized into *chains* in which the target version of one delta is the source version of the next delta in the chain. If a delta's target version is newer than its source version in the history, then it is referred to as a *forward delta*. A *reverse delta* has the opposite orientation: an older target version and a newer source version.

A delta-compressed storage system could simply store the first version of a particular file as its literal version and then store every subsequent version in a forward delta chain. A problem with this strategy is

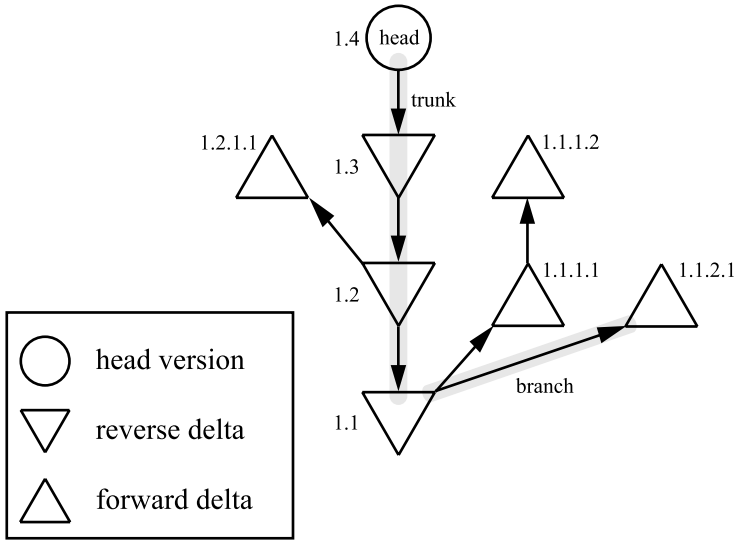

Figure 1: An RCS file showing the placement of forward and reverse deltas. The arrows point from source version to target version. The trunk consists of versions 1.4, 1.3, 1.2, and 1.1, and there are three branches.

that as the number of versions grows, operations must apply long delta chains to access recent versions. Recognizing that recent versions are more likely than older ones to be accessed, RCS instead stores the most recent version (the *head* version) in a distinguished lineage (the *trunk*) of versions. The trunk is made up of of the head version and a reverse delta chain leading back to the original version. Other versions with less distinguished lineage are stored using forward delta chains (known as *branches*), allowing a complete ancestry tree of versions to be stored using a single literal version and both forward and reverse delta chains. An RCS file is illustrated in Figure 1.

An interface for delta-compressed storage should at least support basic insertion and retrieval operations and a facility for labeling versions for external use. Despite its popularity, RCS is not an ideal system for managing delta-compressed storage because its interface is complicated by interfering, unnecessary features of a version-control system, there is little control over its performance, and it makes inefficient use of the file system.

An RCS archive consists of a single file that is managed through ordinary access methods of the underlying file system. The calling application has access to versions stored within an archive through a command line interface that allows you to commit new versions, retrieve existing ones, apply symbolic labels, and perform locking functions.

RCS automatically labels each version corresponding to its position in the ancestry tree. The calling application can select the next allocated version label by acquiring a lock on the parent version. Thus, it usually requires two commands to insert a specific version: one to select (and lock) a parent version and then one to insert the new child version. It is not necessary to pre-select the parent version if the calling program already has a list of its existing children, but that too takes an extra command to acquire.

A version's position in the ancestry tree affects the insertion and retrieval performance of all it's descendents. A version on the trunk has the best performance, whereas performance for a version on a branch

degrades as the length of the trunk increases, since every new reverse delta lengthens the chain between it and the head version. Selecting the parent version prior to insertion thus has two inseparable effects on time and space performance. It provides a non-uniform time–space tradeoff as versions can be placed on or off the trunk, and it provides a hint for compression, making an assumption, which is later tested (§6.1), that the best delta compression will occur when a version-control system computes deltas between immediate relatives. The time–space tradeoff and the compression hint are inextricably linked by the version labeling scheme. In spite of this, choosing version labels requires future knowledge to provide the best performance, and it is a difficult decision for a tool to make. This unnecessary complexity can be passed upward through the higher level version-control system to the user (as CVS does) or it can be ignored, sacrificing performance for the sake of simplicity (as PRCS does).

RCS stores all versions inside a single file using its own file format, and it uses the `rename` system call as an atomic update operation. As a consequence it must rewrite the entire archive during every insertion or administrative update, no matter how small. This is inefficient behavior; it is possible using database techniques to achieve the same atomic update and force far less disk activity. Thus, even simple RCS operations are highly disk bound, and update performance degrades as the size of the archive grows.

2.1 XDFS — The Xdelta File System

The Xdelta File System (XDFS) is designed to improve upon RCS strictly for the purpose of managing delta-compressed storage. XDFS is not a version-control system, but it is intended to support one. The principle goals for XDFS are to support efficient operations, independently of total archive size, and to provide a simple interface for application programming. It accomplishes this by the use of several compression strategies with varying time–space performance, eliminating the use of ancestry and compression hints entirely, and through implementation techniques, including the use of a separate interface for labeling versions and the use of transactions to enable an efficient atomic update without rewriting its entire archive. Implementation techniques will not be discussed until section 5.

XDFS uses a single lineage for version storage, rather than an ancestry tree; it assigns a sequential version number that labels versions in their insertion order, interleaving all branches without the use of any compression hints. XDFS has two compression methods based on the use of forward and reverse deltas that offer various levels of compression and retrieval performance. We shall use the following definitions to describe them. The literal, uncompressed representation of the i^{th} version is denoted V_i. XDFS stores the sequence of versions $V_1, V_2, ...V_N$ using a combination of literal versions and deltas, where a delta from the i^{th} to the j^{th} version is denoted $\Delta_{(V_i,V_j)}$. An archive consists of a number of *clusters*, each of which contains a single literal version and a connected set of deltas. When discussing a specific XDFS compression method, *XDFS-r* denotes the use of reverse deltas and *XDFS-f* the use of forward deltas.

The forward delta-compression method, XDFS-f, uses the *version jumping* scheme by Burns and Long [6]. Each cluster has a fixed literal version, the *reference version*, which serves as the source for all deltas in the cluster. Consider a sequence of versions with two clusters containing versions V_1–V_{i-1} and V_i–V_N, respectively. Using XDFS-f, the sequence is represented by:

$$\underbrace{V_1, \Delta_{(V_1,V_2)}, \Delta_{(V_1,V_3)}, ...\Delta_{(V_1,V_{i-1})}}_{\text{first cluster}}, \underbrace{V_i, \Delta_{(V_i,V_{i+1})}, \Delta_{(V_i,V_{i+2})}, ...\Delta_{(V_i,V_N)}}_{\text{second cluster}}$$

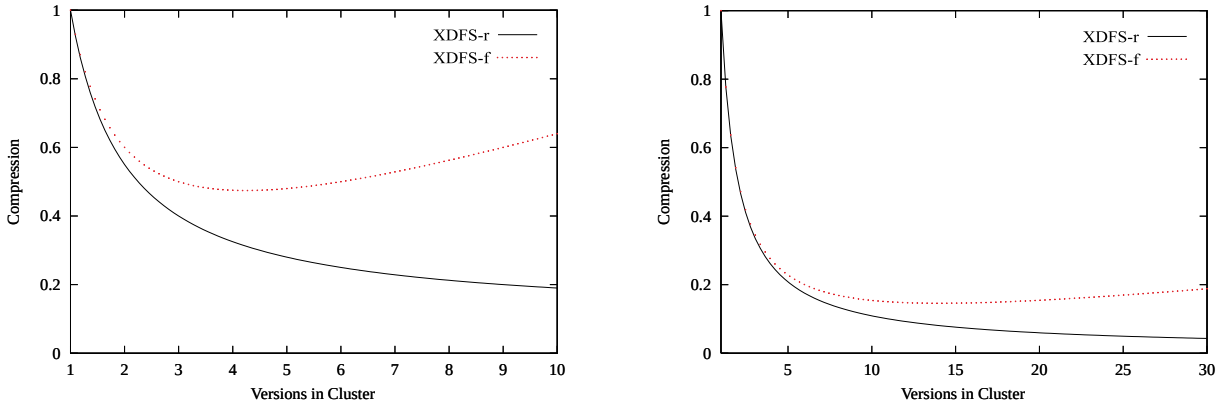

Figure 2: The worst-case cluster compression of XDFS-f and XDFS-r as a function of cluster size, parametrized by the individual delta compression. The left graph has $\alpha = 0.1$ and the right graph has $\alpha = 0.01$.

The reference version is the first version in an XDFS-f cluster, and chain length never exceeds one. Retrieval from an XDFS-f archive is fast because at most one delta must be applied to reconstruct any version. The disadvantage of this technique is that deltas are not computed between adjacent versions, so compression may suffer. After introducing the second compression method, we shall see by how much.

The reverse delta-compression method, XDFS-r, uses a chain of reverse deltas much like the RCS trunk. Its reference version, like the RCS head version, changes with every insertion, and it stores a single chain of reverse deltas to reconstruct the other versions. This technique has better compression than XDFS-f because deltas are computed between adjacent versions, but it has slower retrieval because it must apply more than one delta to reconstruct past versions. This is how XDFS-r represents the same version sequence as above:

$$\underbrace{\Delta_{(V_2,V_1)}, \Delta_{(V_3,V_2)}, \ldots \Delta_{(V_{i-1},V_{i-2})}, V_{i-1}}_{\text{first cluster}}, \underbrace{\Delta_{(V_{i+1},V_i)}, \Delta_{(V_{i+2},V_{i+1})}, \ldots \Delta_{(V_N,V_{N-1})}, V_N}_{\text{second cluster}}$$

An XDFS-r cluster sustains good compression but the delta chain grows longer with each insertion, increasing retrieval time for past versions, whereas an XDFS-f cluster sustains good retrieval time but its compression gradually degrades. For these separate reasons, both methods limit the size of a cluster. XDFS-r simply has a configuration parameter to limit chain length, the *maximum versions per cluster*, that bounds retrieval time for versions in the cluster.

As an XDFS-f cluster grows, changes between the reference version and the most recent version accumulate, leading to larger deltas. Using worst-case analysis, Burns and Long give an upper bound for the cluster compression as a function of individual delta compression α [6]. Their results are reproduced in the graphs of Figure 2 for two values of α. Whereas XDFS-r compression continues to decrease, approaching the limit α, XDFS-f compression reaches a minimum and then begins to rise. XDFS-f limits the size of a cluster by ceasing when it finds a local minimum in compression.

There is a potential problem with interleaving branches in a single lineage caused when the content of two or more branches diverge from one another. In this case, deltas will be computed between increasingly

unrelated versions every time a delta crosses between branches. XDFS-r features an additional compression technique called the *source buffer* designed to alleviate this problem. The source buffer retains a number of additional compression sources taken from recently computed deltas, allowing one delta to copy directly from another delta instead of duplicating that content. A delta that is computed between independently evolving branches is cumulative—it reflects the sum of their mutual evolution. The rationale behind the source buffer technique is that diverging branches may benefit from the ability to copy from previously computed deltas since branch-crossing deltas are likely to include portions of previously computed branch-crossing deltas. Although this technique may be effective, automatic detection and handling of diverging content should best be handled at a higher level, using some form of indirection.

The retrieval function in XDFS is independent of delta orientation—the same operations are performed using XDFS-r and XDFS-f. Given a target version and the chain of deltas that produces it, XDFS then generates an index mapping byte ranges in the target version to immediately available byte ranges stored in the system. This intermediate representation, known as the *current index*, is constructed by reading every delta in the chain, one at a time, and translating the copied ranges. After the current index has been computed, pages of the reconstructed version are filled in, one at a time, as the application reads them.

The number of instructions in the intermediate representation tends to increase as the delta chain length grows, but the number of instructions in any representation is bounded by the length of the target version because it requires at most one instruction per byte of reconstructed content. Assuming that there are n deltas in the chain and that the maximum number of instructions encountered during reconstruction is z, which is a measure of the number of changes accumulated along the chain, the reconstruction algorithm runs in time $O(nz \lg z)$.

2.2 Related Work

A related approach to the ones presented here is used by database systems and text editors to maintain a history of modifications to a database object or text file. The EXODUS storage manager [7] and the SDS2 software development system [1] both use a B-tree representation for storing large, versioned objects and allow sub-tree sharing to achieve storage efficiency. Neither system has any published experimental storage efficiency results.

A similar but more sophisticated approach by Fraser and Myers [14] uses a generalization of AVL trees to store modifications so that all versions can be reconstructed in an in-order pass through the tree. A system by Yu and Rosenkrantz [55] stores literal versions and deltas similar to RCS and focuses on achieving linear-time reconstruction by storing a separate index for every version (this amounts to storing the XDFS current index for every version). Neither system is oriented towards efficient secondary storage, nor do they have any published experimental storage efficiency results.

3 Delta Compression Algorithms

This section is intended to give a brief introduction to the Xdelta algorithm and some of the issues that distinguish it and various other delta algorithms. Delta algorithms fall into two classes depending on the type of algorithm used to compute a delta. The *copy/insert* class of delta algorithms use a string matching

technique to locate matching offsets in the source and target versions and then emit a sequence of *copy* instructions for each matching range and *insert* instructions to cover the unmatched regions. The *insert/delete* class of delta algorithms use a longest common subsequence (LCS) algorithm, or equivalently a shortest edit distance algorithm, to compute an edit script that modifies the source version into the target version. A commonly used program for generating deltas is `diff` [31], which belongs to the insert/delete class of delta algorithms. It is also used by RCS.

Insert/delete algorithms are less appropriate than copy/insert algorithms for storage, transmission, and other mechanical purposes, although they are more appropriate for human viewing. Insert/delete algorithms have inherently greater time complexity than the typical greedy copy/insert algorithm [49], so in practice programs like `diff` break their input into line-delimited records to reduce input size. This technique may be a reasonable approximation for text inputs, but it performs poorly in general on binary files. One study has already demonstrated copy/insert algorithms that perform well on binary files and also outperform `diff` on text files [25].

Insert/delete algorithms do not produce optimal deltas because they weigh an insert instruction the same as a delete instruction, despite the fact that insert instructions consume more space. An insert/delete edit script transforming "proxy cache" into "cache proxy" will contain six deletions and six insertions, twelve instructions in all, whereas a copy/insert delta can do the same in just three instructions: (1) copy "cache", (2) insert a space character, and (3) copy "proxy". Insert/delete algorithms do not consider the high level structure of the inputs; based on this, Lopresti and Tomkins present several refined block edit cost models with applications to such things as molecular biology and handwriting analysis [28].

The most basic copy/insert algorithm is a greedy one. In a single pass through the target version, it picks the longest available match at any given offset and then continues searching at the end of the match. By assuming a simple but unrealistic cost model for delta encoding, the greedy algorithm can be proven to have optimal compression [5]. An implementation of the greedy algorithm must choose a technique for finding the longest matching strings between versions.

For discussing algorithmic complexity, we'll use M as the source version length and N as the target version length. One simple, constant-space approach that uses no auxiliary data structure at all resorts to a linear-time search for every lookup operation; the resulting algorithm has time complexity $O(MN)$. The bdiff algorithm uses a suffix tree instead [49], which can be constructed in time and space proportional to the source version length, resulting in an algorithm with time complexity $O(N + M)$ and space complexity $O(N)$.

Although the suffix-tree approach to string matching is optimal (using the simplified cost model), it has several disadvantages. It requires linear space, which can be prohibitive for large files, and in addition it has a fairly large constant factor because it is a pointer-rich data structure. The suffix-tree approach also tends not to be as fast as an alternative implementation based on hashing techniques [15].

The common alternative to suffix-trees is to use a hash function (or *fingerprint*) to identify potential string matches. Based on the Rabin-Karp string matching algorithm [10], which uses an incremental string hashing function, these techniques generally compute a hash table of fingerprints (the *fingerprint table*) for substrings in the source version and then use a sliding window to incrementally compute the fingerprint value at every offset in the target version[1]. As the greedy algorithm progresses, it searches for each target

[1]There are two hash functions present, an incremental one for computing fingerprints and an ordinary one for hashing finger-

fingerprint in the fingerprint table and uses the result to find the longest match.

There are several approximations to the optimal algorithm that must be made with this approach. First, the fingerprint function has an associated width, denoted s, that determines the minimum length of any match that can be located, and thus the minimum length of any copy instruction. This is a sensible modification in any case, since short copy instructions are likely to cost more than their equivalent insert instructions. The second departure is made in dealing with hash collisions in the fingerprint table. There are three causes for of a hash collision; one may occur when (1) the fingerprint function collides, (2) the hash function used in the fingerprint table collides, and (3) the source version contains identical substrings. The most prevalent of these conditions is the last (e.g., when a file is zero-padded to preserve page alignment). If each of the identical substrings is placed in the fingerprint table (using hash chaining) then the search to identify a longest match will, in the worse case, take linear time, producing a total time complexity of $O(MN)$ [5], an undesirable result. A solution to the collision problem presented by Burns is to use a *next match* policy, where the first match that appears after the last encoded match is taken [5]. This strategy depends on the assumption that matches are ordered within the source version, which holds for insertions and deletions, but not for block moves.

Aside from dealing with hash collisions, improvements to the basic greedy algorithm are generally concerned with running in constant space so that they can operate on large files. The most basic form of this, used by bdiff, is to divide each version into blocks along fixed boundaries and run a linear-space algorithm on each corresponding pair of blocks. More advanced algorithms attempt to synchronize a pair of fixed-size windows as they pass through each version. Burns presents two interesting techniques that work along these lines [5].

Text compression and delta compression are complementary techniques; a system would likely post-compress deltas using a traditional text compression algorithm (RCS does not). Vo's Vdelta is a constant-space, linear-time algorithm that combines traditional text compression techniques with delta compression [25]. Separating text compression from delta compression benefits in reduced complexity since it is possible to use off-the-shelf text compression tools, and the XDFS source buffer technique also relies on the use of a plaintext patch file. For these reasons the Xdelta system does not directly incorporate text compression into its algorithms.

To summarize, there are several different types of copy/insert algorithm available. The greedy algorithm is optimal and parametrized by a string matching function and its corresponding data structure; algorithms are available that are constant-space, quadratic-time or linear-space, linear-time. A class of approximations to the linear-space, linear-time greedy algorithm uses hashing techniques for efficiency, although these alone do not improve algorithmic complexity. Another class of approximations uses less complete matching information, achieving constant-space and linear-time.

3.1 The Xdelta Algorithm

The Xdelta algorithm is an approximation of the greedy algorithm based on the hashing techniques already described. It is a linear-time, linear-space algorithm, and was designed to be as fast as possible at the cost of sub-optimal compression. It uses the same fingerprint function as used by gzip, adler32 [13], with a width of $s = 16$ bytes. Pseudo-code for the basic Xdelta algorithm is given in Figure 3. The `computeDelta`

print values into buckets of the fingerprint table.

function implements the basic greedy algorithm, making a calls to `initMatch` to build a string matching data structure for the source version, `findMatch` to find the longest match at a particular offset in the target version, and `outputInst` when it generates an instruction.

A key feature of the algorithm is the manner in which it deals with hash collisions. The Xdelta `initMatch` function builds a hash table mapping fingerprint values to their offsets for blocks of size s in the source version. There is only one entry per bucket in the fingerprint table, and a hash collision always *clobbers* the existing entry. After populating the fingerprint table, it processes the target version in *pages*, fixed-size blocks with size determined by the underlying I/O subsystem. The `findMatch` function searches for a matching fingerprint in the hash table and then uses direct string comparison to check whether a match exists, also extending the match as far as possible.

Many details are omitted in the pseudo-code. For example, the direct string comparison actually extends the match as far as possible in both directions, with the exception that it will not back-up past a page boundary (to support stream-processing). Fingerprints are inserted into the fingerprint table in the reverse order that they appear in the source version, giving preference to earlier (potentially longer) matches.

The decision to use a linear-space algorithm is justified as follows. First, the constant is quite small since the algorithm uses no pointers and only one 32-bit word of storage per entry in the fingerprint table. The fingerprint table is constructed with a number of buckets b equal to a prime number such that $\lfloor \frac{N}{s} \rfloor < b \leq 2\lfloor \frac{N}{s} \rfloor$. At four bytes per bucket, the space used for string matching in Xdelta is bounded by $\frac{N}{2}$ bytes. This means that the algorithm should be able to handle any files that have been modified in a text editor, since an editor requires more space. Most files fit into main memory, and it did not seem worthwhile to sacrifice compression performance on the vast majority of files without data to characterize the large ones. Aside from that, many large files fall into categories that do not delta compress well in the first place, such as:

- Files with a compressed encoding. The use of text or lossy compression techniques within a file often prevent it from achieving good delta compression, since small changes in uncompressed content can cause dramatic changes in the compressed content.
- Image data, which is two dimensional. The delta compression algorithms presented here only apply well to one-dimensional data. Motion-video encodings provide a lossy form of delta compression for use on images and are more appropriate.
- Database files. A database system will usually use a log for recovery purposes, and the log contains a list of changes so the need to compute deltas on these files can be avoided using application-specific techniques.

Xdelta supports delta compression using multiple source inputs, a feature that supports the XDFS source buffer technique. Whereas the delta compression algorithms discussed up until this point operate on a pair of versions, Xdelta accepts up to $s - 1$ source inputs in addition to the target version.

Xdelta uses a split encoding that separates the sequence of instructions (the *control*) from the data output by insert instructions (the *patch*). There are several algorithms that operate only on control data, so the encoding is split to avoid reading unnecessary patch data during these computations. The control contains a header listing all of the files referenced by copy instructions within, so called *copy sources.* A copy instruction simply refers to a copy source by providing an index into the table. To avoid special cases, the patch is treated as an ordinary file; in fact there is no no insert instruction—all instructions are copies.

```
computeDelta(src, tgt)
  i ← 0
  sindex ← initMatch(src)                    ▷ Initialize string matching.
  while(i < size(tgt))                       ▷ Loop over target offsets.
    (o, l) ← findMatch(src, sindex, tgt, i)  ▷ Find longest match.
    if(l < s)
      outputInst({insert tgt[i] })           ▷ Insert instruction.
    else
      outputInst({copy o l })                ▷ Copy instruction.
    i ← i + 1

initMatch(src)
  i ← 0
  sindex ← empty                             ▷ Initialize output array (hash table).
  while(i + s ≤ size(src))                   ▷ Loop over source blocks.
    f ← adler32(src, i, i + s)               ▷ Compute fingerprint.
    sindex[hash(f)] ← i                      ▷ Enter in table.
    i ← i + s
  return(sindex)

findMatch(src, sindex, tgt, o_tgt)
  f ← adler32(tgt, o_tgt, o_tgt + s)         ▷ Compute fingerprint.
  if(sindex[hash(f)] = nil)
    return(−1, −1)                           ▷ No match found.
  o_src ← sindex[hash(f)]
  l ← matchLength(tgt, o_tgt, src, o_src)    ▷ Compute match length.
  return(o_src, l)
```

Figure 3: Pseudo-code for the Xdelta algorithm. The `computeDelta` function implements the basic greedy copy/insert algorithm, accepting a source and target version as inputs. The `initMatch` and `findMatch` functions perform string matching using a hash table of fingerprint values for source blocks of length s, the fingerprint width.

4 Delta-Compressed Transport

The most important application of delta compression today is for the efficient use of network bandwidth. It is often possible to significantly reduce the transmitted size of a file by sending a delta relative to a previous version instead. The potential benefit for HTTP transport has already been established [34], and the benefits are inherent to systems that also benefit from delta compressed storage. Despite this potential, designing a distributed system to benefit from delta-compressed transport is naturally more difficult than the task of delta compression alone. There may be a number of different end components participating in such a system—clients, servers, and intermediate proxies—each with a unique set of requirements to be met with limited operating resources. The system may be decentralized; that is, it may (purposely) lack global coordination over its names, identifiers, and policies. System design may be also influenced by the expected workload, such as the upload–download ratio, version update frequencies, or number of times a particular version is requested.

For delta compression to be effective, the participants must store some number of previous versions. Each must locally decide which, and how many versions to store, although some applications will retain all versions for other uses. In the context of delta-compressed transport, the *reference version* denotes the source version used in computing a delta for transport—it is common to both the sender and the recipient. Selection and management of reference versions are critical to an effective delta-compressed transport protocol.

To accommodate delta-compressed transport, a protocol must provide means for associating and uniquely identifying versions. A *family identifier* associates related versions together as a family, indicating a potential for effective delta compression, and a *version identifier* uniquely labels versions within a family. The family identifier is simply assigned when its first version is created and must be globally unique. To provide decentralization, the assignment of version identifiers should be autonomous; there are at least two ways to go about this. The version identifier may consist of a globally unique source identifier (e.g., a DNS host name) followed by a locally unique version label (e.g., a name or sequence number). The version identifier may instead consist of a content-specific identifier (e.g., a message digest). The use of a content-specific version identifier, although probabilistic, guarantees that all participants assign a unique mapping.

4.1 Related Work

There have been several attempts to include delta compression in the HTTP/1.1 protocol, including the HTTP Distribution and Replication Protocol [53] and more recently an IETF draft covering protocol modifications to support delta encoding [32]. The specification refers to the family identifier as a *uniqueness scope,* uses the existing HTTP entity tag for version identification, and refers to the reference version as a *base instance.*

The uniqueness scope is not defined as a single Uniform Resource Identifier (URI), instead it allows for one uniqueness scope to contain a set of URIs. This is known as “clustering”, and it is particularly useful for including the results of a dynamic query in the same family. In a typical scenario, the cluster would be defined as the URI prefix up to the first ‘?’, which commonly signifies the beginning of an argument string. There is also a mechanism for specifying that a document is related to a particular template, a sort of explicit clustering mechanism.

The IETF draft, being only a protocol specification, does not discuss implementation issues such as proxy configuration and version storage. Reference version selection is assisted with a cache-control hint allowing a server to indicate which versions should be retained by the client and for how long. The specification also suggest the use of instance digests for integrity checking [33] and VCDIFF, a proposed communication format based on Vdelta, for delta encoding [27].

A system by Chan and Woo [8] uses two techniques to optimize web transfer. It has no conceptual family identifier; instead it uses a selection algorithm for choosing objects from a cache that are likely to be similar based on a measure of path structure similarity. This is based on the observation that web sites often use a consistent design style for nearby pages, which produces similar formatting structure. The selection algorithm is used to choose some number of similar objects for use as reference versions in computing a delta. Much like the XDFS source buffer technique, it uses multiple inputs for delta compression.

Rsync [52] takes a unique approach to delta compression. It allows a client to request changes to a file from the server without requiring the server to maintain any old versions. Instead it dynamically computes the changes in a two-phase interactive protocol. First, the client divides its version up into a number of evenly sized blocks (with default size 700 bytes) and sends a pair of checksums for each block to the server, one an adler32 checksum and the other an MD5 checksum. The server then does an incremental scan of its version looking for matching adler32 checksum values and, when it finds a match, it verifies that the MD5 checksums also match. Once the server has identified which blocks of its version the client already has, it sends any missing portions along with instructions for reconstructing the new version. The Xdelta algorithm is based on the original Rsync algorithm.

An Rsync HTTP proxy called rproxy has recently been developed [51]. Using the Rsync algorithm to exchange delta compressed content frees both the client and the server from managing reference versions, thus simplifying proxy-cache maintenance. The client maintains a copy of its most recent response and the server simply uses the current version. Rproxy implements clustering, as described above, making this solution is especially appealing for dynamic web content distribution.

CVSup [38] is a distribution and replication system designed expressly for CVS, but in general it is useful for synchronizing sets of RCS files across the network. Its primary user base is the FreeBSD project, and thus it is most commonly used with the FreeBSD data set used for the following experiments. CVSup operates by comparing two instances of an RCS file, a master copy and a replica, and it brings the replica up to date by sending only the missing deltas. Needless to say, this is an extremely effective technique, but it is not completely general because it relies on the fact that RCS and consequently CVS are inherently centralized. This is another consequence of the RCS version labeling scheme—they are treated as global names and thus require centralized control.

Another application of delta-compressed transport is for incremental backup systems. Whereas HTTP, Rsync, and CVSup were mostly designed to support delta-compressed client downloads, a distributed backup system experiences the opposite load—clients send deltas rather than receive them. Instead of sending a whole copy of a new version for backup, the client can send a (forward) delta allowing the server to reconstruct its current version. Burns and Long developed their version jumping technique with exactly this scenario in mind [6], because forward deltas can be immediately stored without the need for server processing, whereas storing reverse deltas requires the server to read the old version first to compute a new delta.

Perhaps the most common (though unreliable) form of delta-compressed transport is the use of `diff` and

`patch`. This technique is subject to a number of common problems, including damaged patches, loss of timestamp information (often to the confusion of `make`), version naming issues, and patch reversal, to name a few, but it does have one advantage not supported by any of the above techniques, which is that it supports the "fuzzy" application of patches. Fuzzy patching allows a delta to be applied even when the source version has been modified, but it can only be done using a special form of insert/delete delta that includes additional context information known as a *context diff*, and it is still error-prone.

4.2 XDFS Delta Transport

Although XDFS is primarily concerned with delta-compressed storage, it includes one essential feature for implementing delta-compressed transport. XDFS has the ability to extract deltas directly from storage, like CVSup, but in addition it can extract a delta between an arbitrary pair of versions—CVSup can only extract the RCS-encoded set of deltas. This *extract* feature automatically combines and inverts deltas as necessary to compute a delta between the requested source and target versions. The operation has the same time complexity as version retrieval, and neither version is ever literally reconstructed.

By supporting delta extraction between arbitrary versions, XDFS is capable of supporting delta-compressed transport for decentralized systems. A version-control system could use XDFS to replicate remote repository content and store local versions—simultaneously—and still benefit from delta-compressed storage and transport. This is in contrast to a system like CVS, which only allows the user to deposit versions in a central, shared repository.

There is one subtle extension to the storage system needed to prevent delta chains from being broken. When a cluster can no longer accept new versions, a new one is begun with the insertion of a new literal version. No deltas are necessary in this case, but without one there is no delta chain to connect the two adjacent versions, preventing the extraction algorithm from operating across cluster boundaries. To remedy this, a special delta is inserted between the final literal version of the old cluster and the first version of the new cluster. This delta includes only the control portion of the delta and no patch; instead it copies what would have been patch content directly from the literal copy of itself. This insures that there is a delta chain connecting all versions in the archive, regardless of the cluster they reside in.

A prototype system named xProxy has been implemented that uses XDFS to implement delta-compressed HTTP transport [12]. Experience with this system confirms that the extraction interface is sufficient for supporting delta-compressed transport. Delta-compression is handled transparently, allowing a system implementor to focus directly on protocol design issues.

5 XDFS — An Application-Level File System

This section describes the XDFS implementation, its interface, and the rationale behind it. It covers how transactions are used to improve file system performance, how the use of transactions also benefits in extensibility, and how a separate interface is used for naming versions without interaction with the delta-compression module. I have implemented this using a portable, application-level file system layer.

At a high level, the file system interface consists of methods for accessing generic, persistent state. The two

fundamental abstractions of a file system interface are files (byte arrays) and directories (name spaces). The XDFS file system layer presents a collection of *nodes*, each of which has a unique *node identifier* (the Unix operating system uses the term *inode* to describe file system nodes). Nodes contain a mapping from strings to *minor nodes* of several types, including: immutable byte arrays, called *file segments*; node identifiers, called *reference links*; and directories, which contain a mapping from strings to node identifiers, the entries of which are called *directory links*. Each node defines a default contents—the mapping of the empty string. A node may be thought of as a file or directory in an ordinary file system. Its map structure allows one to associate arbitrary metadata (secondary attributes) with each file or directory.

The traditional effect of writing into a file is accomplished by replacing the value of a minor node (when that value is a byte array) with a complete new contents, as an atomic operation. Writing a file is not atomic in the traditional file system, instead the `rename` operation allows the contents of a directory link to be atomically modified. This causes problems, however, because it prevents multiple directory links (the Unix operating system uses the term *hard links*) from referring to the same file before and after an atomic operation. The use of an atomic replacement operation makes multiple directory links safe for use.

Since it is possible to safely use multiple directory links, the XDFS directory interface doubles as an interface for secondary indexing; this would not be possible without an atomic replacement operation. To make this efficient, a directory can be selected from three available types: a (1) B+tree that supports ordered retrieval and range-search, a (2) hash table supporting equality search, and a (3) *sequence directory* that stores a record-numbered sequence of entries. B+tree and hash table directories can also be configured to allow duplicate entries. Traditionally, file systems use a string representation for path names, causing character set limitations; paths in XDFS are instead a hierarchy of (8-bit clean) keys without any practical restrictions.

XDFS has transaction support, providing atomicity for complex sequences of operations. This allows several directory updates or file replacements to be considered part of the same logical operation, making it possible to store consistent on-disk structures using XDFS. Because of transactions, the delta compression module can insert a version by making several related updates, whereas RCS is forced to rewrite its entire archive and then rename it, causing more disk activity than necessary. Transactions can improve application performance in this manner, by enabling more efficient application behavior.

Transactions also enable extensibility. For example, a file system stacking interface (as described by Heidemann and Popek [24]) supports interposition of various modular services for extending and evolving the functionality of a file system. As Rosenthal argued, however, lack of transaction support is a major obstacle to file system stacking interfaces because without them it is difficult to support independent failure recovery among composed modules [41].

I have implemented a stacking interface for XDFS using a fourth type of minor node called a *view*. A view represents a file segment for which contents can be reconstructed on the fly. The stacking interface is transparent with respect to reads—views are read in the same manner as ordinary file segments—but not with respect to writes. Instead, a view is created with a special call that supplies its length and a callback used to reconstruct the view; write-transparency is regarded as future work (e.g., to implement transparent text compression). To assist in reconstruction, the implementation of a view may use the information stored in related minor nodes and (through reference links) other nodes as well. To allow modules to be composed in the file system, the node mapping includes a *stack identifier*, simply an integer that allows the use of separate name spaces by different modules.

To summarize, a minor node is referenced by the 3-tuple:

minor node reference: [node identifier, stack identifier, name]

The node identifier field refers to the containing node, the stack identifier field supports composing modules, and the name field allows for secondary attributes. Supporting atomic replacement instead of `rename`, the XDFS directory interface also supports secondary indexing. Transactions can improve performance by improving application behavior, and they enable extensibility by allowing for independent failure recovery.

5.1 Implementation

XDFS is implemented on top of the Berkeley Database (Berkeley DB) [37], which provides (nested) transaction support and a number of other useful features. XDFS uses Berkeley DB to store its nodes, directories, and small file segments, and it uses the underlying file system for bulk data storage. Similar to a journaling file system, XDFS benefits from database logging techniques by the ability to exploit sequential write performance for small operations, retaining the bulk transfer efficiency of the underlying file system for large file segments.

XDFS does not have any explicit control over the placement of data or metadata blocks in the underlying file system, which could limit performance in some applications. The best solution to this problem is for the operating system to expose a better interface that gives application-level file systems more explicit control [26]; still, there are several placement policies that XDFS implements in the interest of improved performance. XDFS stores minor nodes in a B+tree keyed by their minor node reference, maintaining locality between entries related by a common node. Small file segments, defined to be those smaller than a threshold (currently 2048 bytes), are stored in a separate B+tree using the same key as their minor node, maintaining locality between related short file segments. Large file segments and XDFS directories are stored as ordinary files in the underlying file system, split into subdirectories so that no single directory grows to contain too many files.

Figure 4 illustrates an XDFS-r archive containing three versions, represented in XDFS by a *location node.* The location node has several minor nodes: a reference link to the literal version of the current cluster (default), a file recording internal counters and statistics (“state”), and a reference link to a sequence directory which refers to the sequence of versions (“sequence”). Literal versions are represented as ordinary XDFS file segments, and delta-compressed versions are represented using views. A node containing an XDFS view also includes, as secondary attributes, the delta control and reference links to each copy source. In a typical scenario there are two copy sources: the patch source, which is generated during delta compression, and the primary source, the version from which the delta was computed.

XDFS uses a sequence directory (represented by a Berkeley DB Queue, with minimum size 8 KB) to index versions by insertion number. The source buffer technique uses an additional sequence directory to store links to recent patch file segments (although it is never fully utilized, due to the source buffer’s small size). These directories are responsible for a great deal of storage overhead, especially when there are only a few versions in the archive. They also require significant extra disk synchronization to create, slowing the initial XDFS insertion operation. Berkeley DB makes five calls to `fsync` to create a sequence directory (three to the log, two to the file itself).

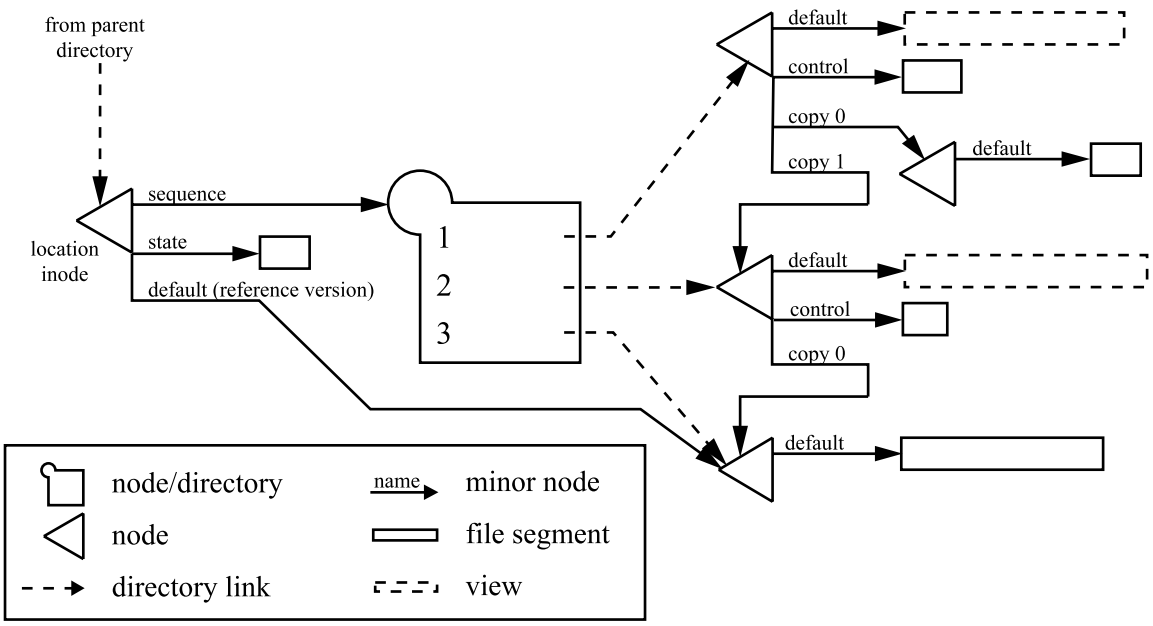

Figure 4: The representation of an XDFS-r archive with one cluster and three versions using XDFS data structures. Version 1, for example, has four minor nodes: the default XDFS reconstruction view, the delta control, an reference link to the patch file (copy 0), and an reference link to version 2, the primary copy source (copy 1).

XDFS calls `fsync` to synchronize any files that have been created in the underlying file system prior to transaction commit. XDFS-f can potentially cause fewer calls to `fsync` than XDFS-r because XDFS-f usually only writes a new delta into the system, whereas XDFS-r replaces a literal version and writes a new delta with each insertion. Since deltas are frequently smaller than the small file threshold, less so than literal files, so XDFS-f has a tendency to make fewer calls to `fsync`.

It is often useful to compute the message digest of a version in conjunction with delta compression for later use in addressing versions. XDFS has an option to automatically compute either the MD5 or SHA-1 message digest of every version—this information is maintained in a secondary index. XDFS assumes that these functions are collision free, and using this assumption it can avoid duplicate insertions. This is provided both for convenience and efficiency—not providing this feature forces an application to make an extra pass through the version prior to insertion.

5.2 Related Work

Stonebraker was the first to point out that the traditional file system interface should do more to assist in the efficient implementation of a database system [45]. The DBMS can generally provide higher level information about disk placement and buffer replacement policies than can be expressed through the file system interface. He states that operating system designers should provide database facilities as lower-level objects and files as higher-level ones. Along these lines, the Inversion File System [36] is built on top of the

POSTGRES [46] extensible database system. The POSTGRES system provides it a number of desirable features, including transactions, concurrency, no-overwrite storage with time-travel, fast crash recovery, and query support for file system data and metadata.

The use of transactions in and over the file system has a long history of study. Standard transaction processing techniques [35] have influenced the designs of both log-structured [40] and journaling file systems [9, 47], but only a few of these services export a transactional interface [36, 23]. It is more common to use transactional techniques as an implementation device providing either efficiency or consistency in a distributed file system [23] or to improve small-write performance [43]. Seltzer studied issues involved in transaction processing with FFS [30] and a log-structured file system [44].

Transactions are critical to operating-system extensibility, since they allow independent failure recovery. Atomic recovery units [20] were studied as a technique for separating file management and disk management using a logical disk abstraction [11]. The MIT Exokernel provides low-level operating system support for application-level file systems [26]. It has been used to improve upon small-file performance by embedding inodes in their containing directory, thus eliminating a level of physical indirection, and using explicit grouping to place small files from the same directory on adjacent disk pages [16]. Extensible transaction support has also been studied in the database community [21, 46, 37, 4].

The first system to use extended, content-based attributes for indexing the file system was the Semantic File System (SFS) [18]. SFS allows conjunctive attribute–value queries to be made using *virtual directories* that store live query results; it emphasizes consistency and supports query result caching. Special *transducer* programs can be registered to automatically extract file attributes. SFS and other systems that support attribute-based file naming [19, 42, 17] all attempt to export a query interface through to the user level. These interfaces must cooperate with the existing semantics, and are idiosyncratic in their use of names to return query results. XDFS instead provides more primitive mechanisms for use in indexing and attributing files.

The Be file system (BFS) [17] provides the ability to store multiple, arbitrary length, named and typed attributes along with a file. Space is reserved in the inode to directly store short attributes, and longer attributes are indirectly stored in an attribute directory. BFS supports automatic indexing on secondary attributes, an expressive query language, and the ability to create live queries. BFS has an unusual feature for returning query results to the application; each inode includes a reference to its containing directory and the name of its entry in that container, allowing BFS to reconstruct the full path name of a file from its inode. Multiple hard links are not supported, as a consequence.

6 Experimental Results

The primary data set used in this study is taken from a large collection of RCS files belonging to the FreeBSD Project's CVS repository, specifically, the CVS repository snapshot included with the June 1999 release of FreeBSD 3.2, USENIX Edition [48], including its source tree, ports collection, documentation, and CVSROOT directory. Table 1 provides statistics describing the RCS files in the FreeBSD CVS data set. The complete data set is listed as *FreeBSD-full*, whereas the set listed as *FreeBSD-min10* includes only

those files with at least 10 versions[2]. The mean ($\overline{x}$) and standard deviation of the mean (σ_x) are listed for a number of per-file statistics.

Better delta compression is achieved when more versions are available to compress, as shown in Figure 2. Figure 5 shows the distribution of versions for the FreeBSD CVS data set. The vast majority of files contain just a few versions—70 percent of files have four versions or fewer. The file with the greatest number of versions is the kernel configuration file `sys/i386/conf/LINT`, which has 712 versions.

Version size also affects the overall performance of a delta compression scheme, due to storage overhead costs. Figure 6 shows the distribution of version sizes for the head version of each file, which is approximately the distribution of file sizes in FreeBSD 3.2. Version sizes have a familiar file system distribution—most versions are small, although most of the volume belongs to large versions [2, 54]. The largest version in the data set is the word list `src/share/dict/web2`, at 2,486,905 bytes.

6.1 Delta Compression

I ran experiments comparing the Xdelta algorithm against GNU `diff` [22] using the FreeBSD-full data set. In the first experiment, the source and target versions used for delta computation are defined by deltas in the input data set, thus measuring the effect of RCS ancestry. The second experiment ignores ancestry and uses source and target versions defined by insertion order, thus measuring the effect of interleaving branches in a single reverse delta chain. As an external program, `diff` incurs the cost of `vfork` and `exec`. Due to this, the time to execute an empty external program is included in the measurements.

The test machine has dual 450 MHz Intel Pentium II Xeon processors running Red Hat Linux 6.1 (kernel 2.2.12). Each testing method is executed four times on each pair of versions, and the total compression time is reported as the sum of the mean execution time for each. The standard deviation of the mean in the reported total is less than 1 percent for each measurement. Table 2 gives the results of these measurements.

Deltas arranged according to the original RCS ancestry are 9–12 percent *larger* than those arranged by insertion order. This shows that, at least for the FreeBSD-full data set, that the use of a single lineage is not significantly harmed by diverging branches. Instead, it shows that versions are more likely to be related to versions created nearby in time than to their immediate ancestor—that the same changes appear on different branches. Clearly the RCS ancestry information does not always make a good compression hint.

Xdelta outperforms `diff` at compression by 33–35 percent, and is also faster. The majority of the difference in compression time, however, comes from the overhead of calling `vfork` and `exec` to execute `diff`. This overhead is not inherent to `diff`, although it is an artifact of the common implementation used by RCS and it is not easily avoided. Subtracting the overhead cost of `vfork` and `exec` from the execution times for `diff`, Xdelta is approximately 30 percent faster. Including that cost, Xdelta is faster by 67 percent.

Figure 7 shows the distribution of delta sizes for Xdelta with deltas arranged by insertion order. The size reported for each delta is the sum of both the control and patch data sizes. Patch data accounts for an average 80 percent of the delta by size. The data indicates that most deltas are small, with 73 percent

[2]Thirty-three files were skipped because comments included in the output of the RCS `rlog` command included the output of *other* `rlog` commands, making them difficult to analyze.

Data Set	FreeBSD-full		FreeBSD-min10	
Total Files	52,570		5,614	
Total Versions	293,907		135,578	
Total Head Version Size (bytes)	384,144,235		77,463,039	
Total Uncompressed Size (bytes)	3,448,857,689		2,342,852,038	
Versions/File	$\overline{x} = 5.59$	$\sigma_x = 12.42$	$\overline{x} = 24.15$	$\sigma_x = 32.09$
Branches/File	$\overline{x} = 0.70$	$\sigma_x = 0.78$	$\overline{x} = 1.82$	$\sigma_x = 1.07$
Forward Deltas/File	$\overline{x} = 1.55$	$\sigma_x = 3.66$	$\overline{x} = 5.05$	$\sigma_x = 10.27$
Reverse Deltas/File	$\overline{x} = 3.04$	$\sigma_x = 9.94$	$\overline{x} = 18.10$	$\sigma_x = 25.50$
Head Version Size (bytes)	$\overline{x} = 7,307$	$\sigma_x = 29,002$	$\overline{x} = 13,798$	$\sigma_x = 55,086$
Uncompressed File Size (bytes)	$\overline{x} = 65,605$	$\sigma_x = 659,144$	$\overline{x} = 417,323$	$\sigma_x = 1,966,048$

Table 1: RCS statistics for the FreeBSD CVS data sets. There are nearly 300,000 versions in the FreeBSD-full data set, which occupy 3.4 gigabytes of uncompressed space. The FreeBSD-min10 data set accounts for 11 percent of the files, 46 percent of the versions, and 68 percent of the volume of uncompressed data. The mean ($\overline{x}$) and standard deviation of the mean (σ_x) are listed for a number of per-file statistics.

Method	Delta Arrangement	Time (sec.)	Delta Size	(bytes)	Total Size
Diff	RCS ancestry	769	$\overline{x} = 544$	$\sigma_x = 6734$	131 MB
Diff	Insertion order	757	$\overline{x} = 498$	$\sigma_x = 6087$	120 MB
Xdelta	RCS ancestry	251	$\overline{x} = 363$	$\sigma_x = 4454$	88 MB
Xdelta	Insertion order	246	$\overline{x} = 326$	$\sigma_x = 3979$	79 MB
Vfork/Exec		407			

Table 2: Delta compression performance, measured on the FreeBSD-full data set. Deltas are either arranged using the original RCS ancestry or ordered by insertion sequence. The vfork/exec entry measures only the time to `vfork` and `exec` an empty program, and shows that `diff` spends more time in startup than it spends computing deltas. The RCS ancestry produces deltas that are 9–12 percent *larger* than those of the simpler insertion sequence.

being smaller than 100 bytes and 95 percent being smaller than 1 KB. The largest delta, at 1,436,782 bytes, belongs to the file `doc/en/handbook/handbook.sgml` and occurred when the entire file was split into sections and replaced instead by a table of contents (deletion between the versions of a reverse delta causes insertion in the delta).

Figure 8 shows the distribution of α for Xdelta with deltas ordered by insertion order. The graph indicates, for example, that 50 percent of deltas have α less than 0.02 and that 80 percent of deltas have α less than 0.12.

6.2 Delta-Compressed Storage

XDFS and RCS are compared in terms of: ideal compression, which measures data stored by the delta-compression algorithm itself; actual compression, which includes file system storage overhead; insertion time; and retrieval time. Both data sets are measured; FreeBSD-min10 is interesting because it features much less overhead than FreeBSD-full, due to its high number of versions per file, and because it emphasizes the asymptotic degradation of RCS insertion performance.

RCS is tested in two configurations: one that preserves the original ancestry tree, denoted *RCS-t*, and one that disregards it in favor of single lineage using insertion order, denoted *RCS-l*. As a basis for comparison, two uncompressed storage methods are also tested: *XDFS-none*, which measures XDFS file system overhead alone, and *FS-none*, which measures the performance of the underlying file system. XDFS-f is tested in its only configuration. XDFS-r is tested with the maximum versions per cluster (MVC) set to 5, 10, 20, and 40 (default) and with a source buffer enabled.

The experiment consists of two passes over the set of input files, first storing every version, followed by retrieving every version. The number of files stored in the underlying file system by each method has an effect on performance, especially storage overhead. All of the methods use a directory tree in the underlying file system for storage: the RCS and FS-none methods duplicate the original RCS directory structure, and XDFS uses an automatically-allocated directory structure. All of the methods except for RCS store more than a one file or directory per input file. FS-none uses one directory per file, inside which it stores one file per version. The XDFS methods use at least one sequence directory, and two when configured with a source buffer.

The experiments were run on the same machine described above (§6.1) using the standard Linux ext2 file system. Three separate Ultra2 SCSI (LVD) disks were used with an Adaptec AIC-7890 disk controller: one to contain the input data set and host operating system (an IBM Ultrastar 9ES, model DDRS-34560D), one for the constructed archives (a Seagate Cheetah, model ST39103LW), and one used only by Berkeley DB for log files (another ST39103LW). The separate log disk, made possible by the Berkeley DB system architecture, is used to optimize for sequential write performance. All methods use `fsync` to synchronize the disk after writing a new file into the underlying file system (RCS was modified to do so), and Berkeley DB uses `fsync` on its log at transaction commit. RCS measurements include the cost of `vfork` and `exec`, and RCS uses an additional command prior to the initial insertion to establish non-strict locking (`rcs -i -U`). The XDFS methods use one transaction per insertion. Measurements do not account for space consumed by the Berkeley DB log or shared memory regions, since these support qualities unavailable in RCS such as recovery and concurrency. Berkeley DB was configured with a 64 KB log buffer, a 1 MB page cache, and its lock manager fully enabled.

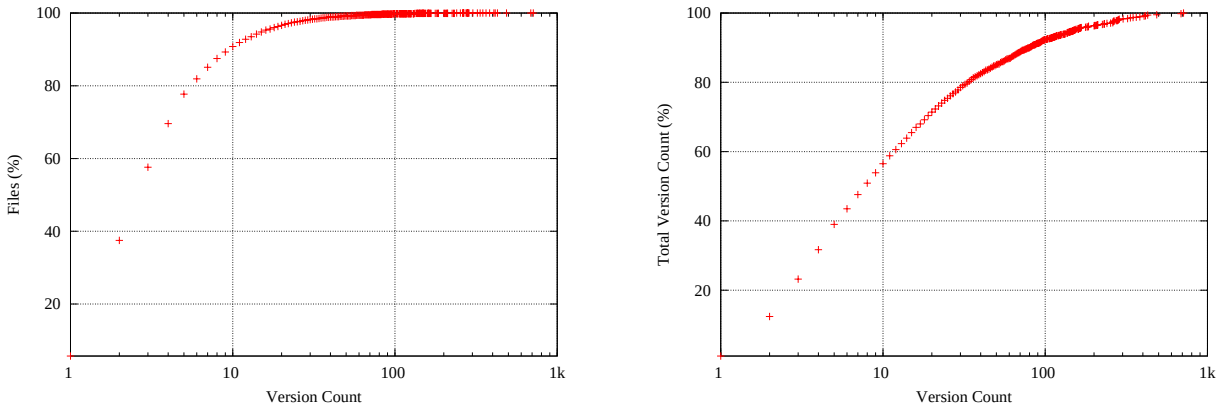

Figure 5: The cumulative distribution of version count for files in the FreeBSD-full data set. The left graph is weighed by the number of files, and the right graph is weighed by the number of versions. For example, the left graph indicates that only 10 percent of files have more than 10 versions, and the right graph indicates that 45 percent of versions belong to that group of files.

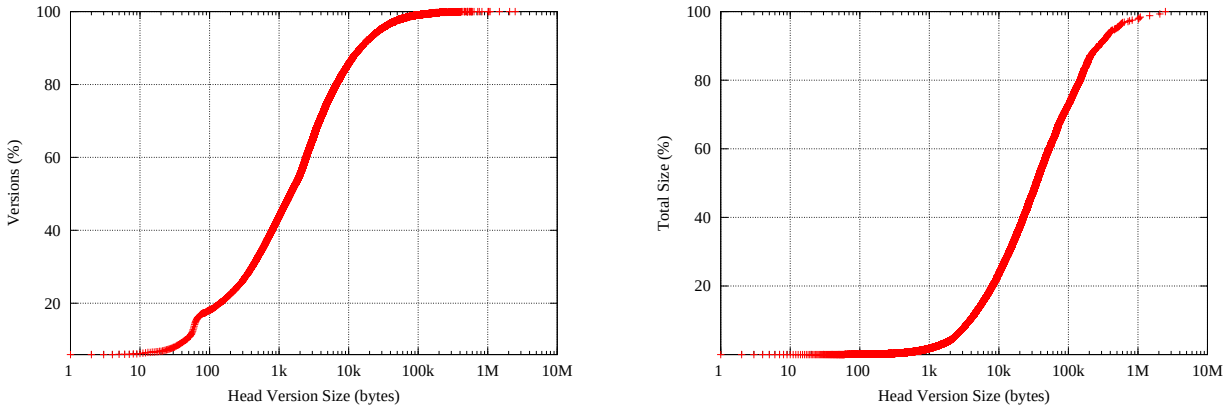

Figure 6: The cumulative distribution of version size for the head version of each file in the FreeBSD-full data set. The left graph is weighed by the number of versions, and the right graph is weighed by the size of each version. For example, the left graph indicates that 15 percent of the head versions are larger than 10 KB, and the right graph indicates that 75 percent of the volume belongs to that group of versions.

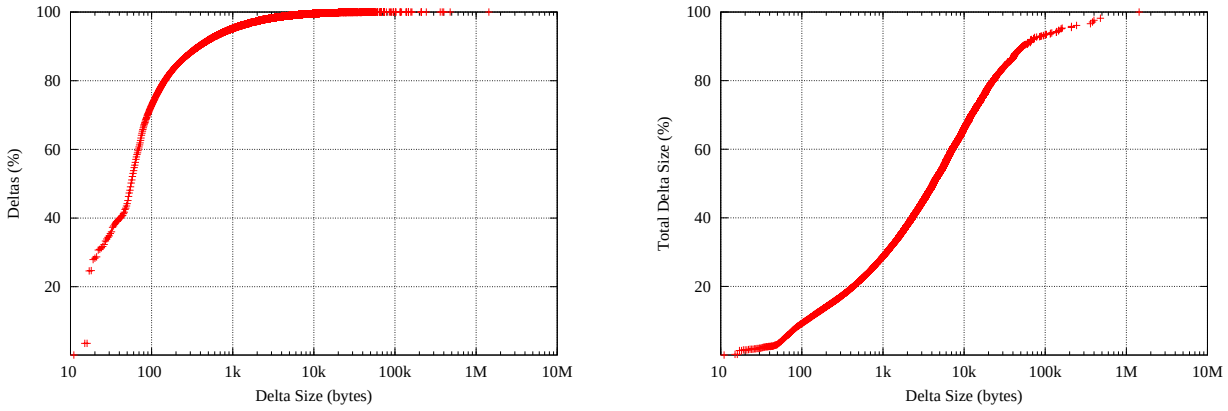

Figure 7: The cumulative distribution of delta sizes for Xdelta with deltas arranged by insertion order. The left graph is weighed by the number of deltas, and the right graph is weighed by the size of each delta. The graph indicates that most deltas are small—73 percent of deltas are smaller than 100 bytes. These sizes account for both the control and patch data of the delta.

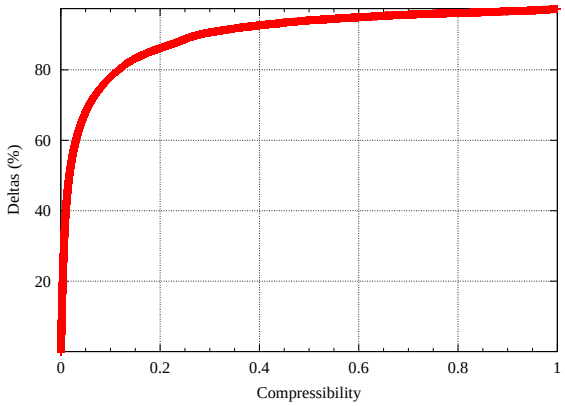

Figure 8: The cumulative distribution of delta compression for Xdelta with deltas arranged by insertion order. The graph indicates that 50 percent of deltas have α less than 0.02 and that 80 percent of deltas have α less than 0.12. Due to overhead in the Xdelta encoding, 2.5 percent of deltas are actually larger than their target version and are not displayed on this graph.

Figure 9 shows insertion time for the FreeBSD-min10 data set as a function of insertion number, normalized by the FS-none method. The graph plots the average normalized insertion time for the n^{th} insertion taken over all files with at least n versions. Results are shown up to 400 insertions, at which point only six files have enough versions. As expected, the insertion time of RCS increases steadily as a function of insertion number due to the cost of rewriting its entire archive at each insertion. The XDFS methods are significantly slower at initial insertion than RCS: the RCS methods are 5.6 times slower than FS-none, whereas the XDFS methods are 16.5–20 times slower than FS-none. XDFS is slower at initial insertion, in part, due to extra disk synchronization needed to create a Berkeley DB database file. Due to an increase in average version size, insertion time for the XDFS methods drops slightly as a function of insertion number: the average 1^{st} version size is 6.6 KB whereas the average 400^{th} version is 53 KB.

Complete timing results for both data sets are shown in Table 3. Due to the time taken to collect a single set of results, I was only able to collect two complete runs, and not all the XDFS methods are tested with the FreeBSD-full data set since they do not reveal much extra information. The mean, uncertainty, and factor relative to FS-none are reported for each timing measurement. Retrieval times are given, but are not as important as insertion times because versions can be cached once they have been retrieved.

Complete compression and storage overhead results are shown in Table 4. The ideal storage performance of the algorithms is given in absolute terms and in terms of compression. Actual storage is measured as the number of disk blocks utilized in the underlying file system, and is expressed in both absolute terms and relative to ideal storage, the overhead storage factor. Actual compression is given as the product of ideal compression and overhead storage.

With the FreeBSD-min10 data set, which has low overhead, XDFS-f and XDFS-r significantly outperform RCS at insertion, by up to 40 percent. With the FreeBSD-full data set, which has high overhead, RCS outperforms XDFS by varying degrees: XDFS-f by 3 percent and XDFS-r by 12 percent. The difference in performance between XDFS-f and XDFS-r is a result of XDFS-f's tendency to make fewer calls to `fsync`.

Varying the maximum versions per cluster parameter of XDFS-r has the desired effect; a higher value of MVC achieves better compression with worse retrieval time. The default setting of 40 versions produces ideal compression that is similar to the RCS methods. As expected, XDFS-r ideal compression is better than XDFS-f: by 60 percent with FreeBSD-min10 and 35 percent with FreeBSD-full.

Storage overhead is a serious concern for the XDFS and FS-none methods. For the non-RCS methods storage overhead is, in most cases, greater than the ideal compressed storage. Actual compression suffers, as a result. With the FreeBSD-full data set, for example, XDFS-r requires 58 percent more actual storage than RCS-l even though its ideal storage is slightly better. This indicates that it is more important to improve file system storage efficiency than to continue to improve delta-compressed storage techniques. On the bright side, XDFS-none has less storage overhead than FS-none. This is a result of the XDFS small file allocation policy.

The source buffer technique improves compression slightly, but at a great cost in both storage overhead and insertion speed. The source buffer was configured to allow up to 14 ($s - 2$) patch files (with minimum size 256 bytes) to be used as additional inputs for delta compression. On the FreeBSD-full data set it reduces ideal storage by 9 MB, compared to XDFS-r, but uses an additional 245 MB overhead storage and takes 28 percent longer at insertion.

Figure 10 shows the distribution of XDFS segment sizes for the FreeBSD-full data set encoded using the

XDFS-f method (including literal, control, and patch segments). The XDFS archive has a greater fraction of small files than appeared in either the Sprite [2] or Windows NT [54] file system studies, for obvious reasons. Fifty-four percent of files are smaller than 100 bytes and 84 percent of files are smaller than 1 KB.

Figure 11 summarizes these results by plotting insertion time versus actual compression for the FreeBSD-min10 data set, showing the relative time and space performance of the storage methods in the case where there are many versions per file.

6.3 Delta-Compressed Transport

To evaluate the XDFS extraction procedure, a second data set, denoted *HTTP-dynamic*, is taken from a collection of seven popular web sites that contain news and other information that changes on a daily basis. Much of the content of these web pages is static, as they are generated using templates. The "index.html" page was collected from each of these sites twice daily at 8:00AM and 8:00PM Pacific Standard Time beginning on January 27, 2000 and ending February 29, 2000, except when the site was down. Table 5 describes each of the sites that was used, the number of versions collected, the average page size, and their total size.

Table 6 gives the compression results for a simulated proxy configuration in which successive versions are extracted as deltas and transferred from one archive to another. These results are not augmented with any additional text compression techniques, as would certainly be desirable in a real system.

For the FreeBSD-full data set, this experiment provides little new information. There is a slight improvement in compression over the results in Table 4 due to the fact that every version is transported as a delta whereas the storage system stores one literal version per cluster.

The results for the HTTP-dynamic data set confirm previously stated results on the potential for delta encoding in HTTP [51] [34] with an average reduction of 75 percent. The spread between forward and reverse compression results is smaller for the HTTP data (2 percent) than for the FreeBSD data (5 percent). These results indicate that the compressible HTTP content is relatively static across many versions of same page (whereas the FreeBSD data is more evolutionary).

7 Summary

Delta compression has important, practical applications, but is difficult to manage. XDFS attempts to isolate the complexity of managing delta-compressed storage and transport by making version labeling independent of delta-compression performance: version labeling uses the file system abstraction, and a separately tunable time–space tradeoff modifies performance. Insertion time performance is independent of total archive size due to the use of transactions. XDFS also isolates the complexity of delta-compressed transport protocol design from the delta-compression mechanisms that support it.

The traditional file system has weak consistency semantics: the only atomic operation provided is `rename`. Lack of transaction support in the traditional file system causes inefficient behavior in applications that require consistency. File system transaction support can improve application performance with the benefit

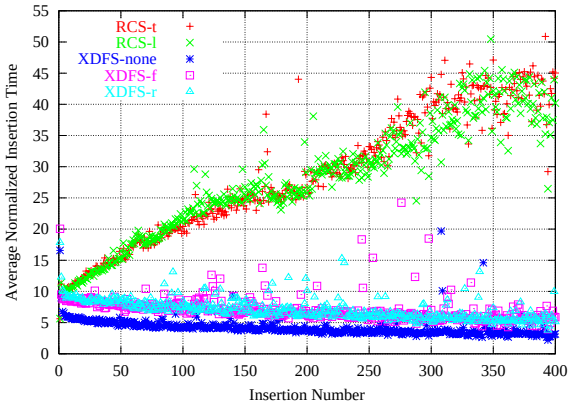

Figure 9: Average normalized insertion time for the FreeBSD-min10 data set as a function of insertion number. The RCS methods' insertion time grows as a function of insertion number because it takes time proportional to the archive size, due to the cost of rewriting the entire archive with every insertion. The XDFS methods are not dependent on total archive size, although they are slower at the initial insertion.

Method	Insertion time	Insertion factor	Retrieval time	Retrieval factor
FS-none	483 sec. ± 0.5%	1.00	315 sec. ± 0.8%	1.00
RCS-t	3129 sec. ± 0.1%	6.47	1169 sec. ± 0.4%	3.71
RCS-l	3134 sec. ± 0.3%	6.48	1235 sec. ± 0.4%	3.92
XDFS-none	1153 sec. ± 1.5%	2.39	453 sec. ± 0.5%	1.44
XDFS-f	1826 sec. ± 0.9%	3.78	803 sec. ± 0.9%	2.55
XDFS-r (mvc=40)	1833 sec. ± 0.2%	3.79	1558 sec. ± 0.4%	4.94
XDFS-r (mvc=20)	1863 sec. ± 1.4%	3.85	1209 sec. ± 1.1%	3.84
XDFS-r (mvc=10)	1874 sec. ± 1.4%	3.88	978 sec. ± 1.1%	3.10
XDFS-r (mvc=5)	1904 sec. ± 1.5%	3.94	851 sec. ± 1.0%	2.70
XDFS-r (srcbuf)	2157 sec. ± 0.7%	4.46	1734 sec. ± 0.8%	5.50

Method	Insertion time	Insertion factor	Retrieval time	Retrieval factor
FS-none	1243 sec. ± 1.3%	1.00	1060 sec. ± 1.9%	1.00
RCS-t	4951 sec. ± 0.1%	3.98	1937 sec. ± 0.2%	1.83
RCS-l	4962 sec. ± 0.3%	3.99	2032 sec. ± 0.7%	1.92
XDFS-none	4000 sec. ± 1.7%	3.22	1302 sec. ± 1.5%	1.23
XDFS-f	5135 sec. ± 1.8%	4.13	2319 sec. ± 5.2%	2.19
XDFS-r (mvc=40)	5669 sec. ± 1.5%	4.56	3371 sec. ± 0.3%	3.18
XDFS-r (srcbuf)	7867 sec. ± 0.3%	6.33	3903 sec. ± 1.7%	3.68

Table 3: Complete timing results for the FreeBSD-min10 (top) and FreeBSD-full (bottom) data sets, taken over two runs. Insertion and retrieval times are the sum of the insertion and retrieval time for individual versions in the data sets. Results are also given as factors relative to FS-none.

Method	Ideal storage	Ideal comp.	Actual storage	Overhead factor	Actual comp.
FS-none	2342 MB	1.000	2777 MB	1.185	1.185
RCS-t	160 MB	0.068	248 MB	1.552	0.106
RCS-l	158 MB	0.068	246 MB	1.555	0.105
XDFS-none	2342 MB	1.000	2685 MB	1.146	1.146
XDFS-f	387 MB	0.165	614 MB	1.589	0.262
XDFS-r (mvc=40)	156 MB	0.067	297 MB	1.906	0.127
XDFS-r (mvc=20)	206 MB	0.088	352 MB	1.710	0.151
XDFS-r (mvc=10)	323 MB	0.138	483 MB	1.493	0.206
XDFS-r (mvc=5)	550 MB	0.235	734 MB	1.335	0.314
XDFS-r (srcbuf)	147 MB	0.063	330 MB	2.233	0.141

Method	Ideal storage	Ideal comp.	Actual storage	Overhead factor	Actual comp.
FS-none	3448 MB	1.000	4520 MB	1.311	1.311
RCS-t	497 MB	0.144	729 MB	1.468	0.212
RCS-l	490 MB	0.142	722 MB	1.472	0.209
XDFS-none	3448 MB	1.000	4420 MB	1.282	1.282
XDFS-f	737 MB	0.214	1498 MB	2.030	0.434
XDFS-r (mvc=40)	479 MB	0.139	1141 MB	2.382	0.331
XDFS-r (srcbuf)	470 MB	0.136	1395 MB	2.967	0.404

Table 4: Complete compression and storage overhead results for the FreeBSD-min10 (top) and FreeBSD-full (bottom) data sets. Ideal storage measures the algorithmic storage requirements, whereas actual storage measure disk utilization. Actual compression is the product of ideal compression and the storage overhead factor.

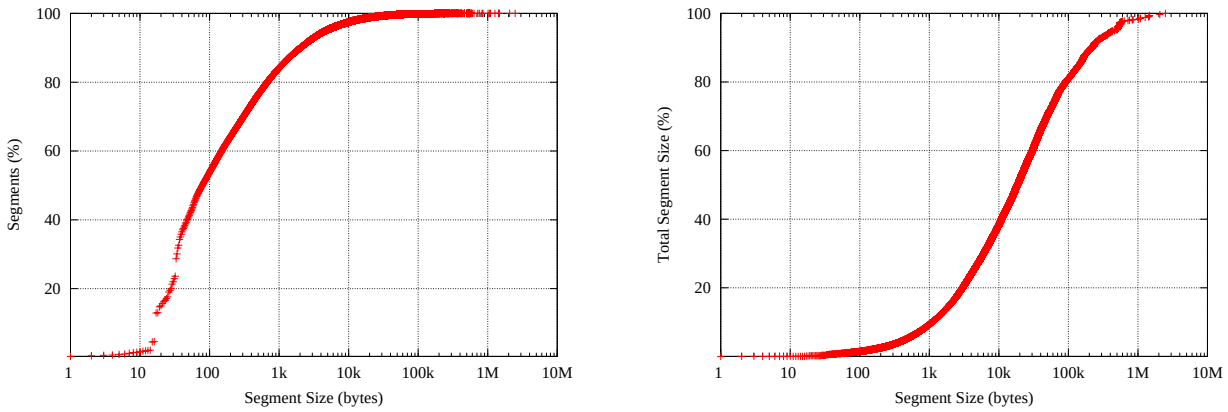

Figure 10: The cumulative distribution of segment size for the FreeBSD-full data set encoded using the XDFS-f method. The left graph is weighed by the number of segments, and the right graph is weighed by the size of each segment.

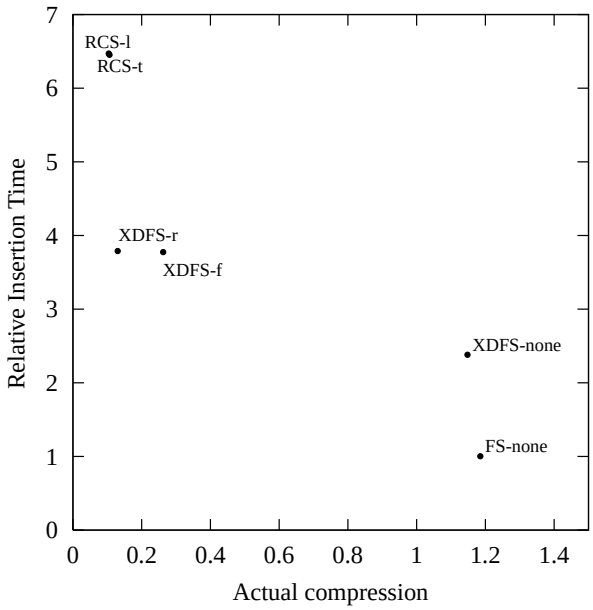

Figure 11: A plot of insertion time versus actual compression for the FreeBSD-min10 data set, summarizing the results in the case where there are many versions per file.

Site	Description	Versions	Average Size	Total Size
cnn.com	CNN's front page news	66	65 KB	4.3 MB
cnnfn.com	CNN's financial news	66	42 KB	2.8 MB
deja.com	Usenet news portal	66	30 KB	2.0 MB
news.cnet.com	CNET's technology news	66	37 KB	2.5 MB
slashdot.org	News for Nerds	65	36 KB	2.4 MB
www.excite.com	Web portal	66	32 KB	2.1 MB
www.zdnet.com/zdnn	ZDNet's technology news	66	59 KB	3.9 MB

Table 5: Popular web sites used for the HTTP-dynamic data set. The site's "index.html" page was collected twice daily for 33 days. Slashdot.org was down during one of the collection periods.

Name	Data set	Compressed Transfer	Uncompressed Transfer	Compression
XDFS-f	FreeBSD-full	657 MB	3449 MB	0.191
XDFS-r	FreeBSD-full	474 MB	3449 MB	0.137
XDFS-f	HTTP-dynamic	5.1 MB	20 MB	0.257
XDFS-r	HTTP-dynamic	4.7 MB	20 MB	0.238

Table 6: Delta compression in simulated proxy configuration for the FreeBSD-full and HTTP-dynamic data sets.

Module	Size (1000 lines)	Files	Function
XDFS	2.2	1	Delta-compressed storage module
DBFS	5.8	6	File system layer
Xdelta	1.6	4	Delta compression algorithm
Common	9.8	21	Input/output routines, object serialization
Experiments	5.5	12	Test and experimental applications
Glib-1.2.6	25.9	60	Open source C utility library
Berkeley DB 3.1.6	115	294	The Berkeley Database

Table 7: Breakdown of program code (mostly C) for the modules comprising XDFS.

of strong consistency semantics. The XDFS file system layer provides an atomic replacement operation as an alternative to `rename`, making the use of multiple directory links safe. As a result, the directory interface can be used for secondary indexing purposes.

The use of transactions permits extensibility by supporting independent failure recovery. The XDFS file system layer's extensibility has been demonstrated through the use of a stacking interface that supports delta-compression. Table 7 shows the breakdown of lines of C code for the modules comprising XDFS. The delta-compression module is self-contained, and the file system layer has a small fraction of the complexity of the underlying database.

Measurements of delta-compression methods over the FreeBSD CVS data indicate that RCS ancestry information is ineffective as a compression hint. Better compression is achieved with the use of a single lineage, indicating that versions on a (version control) branch are more closely related to the current version than they are to their immediate ancestor—diverging branches were not found to be an issue.

Using file system structures to store individual deltas, storage overhead is high. Caused by small files and directories with only a few entries, storage space lost due to fragmentation and file system metadata exceeds the compressed size of the data. At this point, it is more important to improve file system storage overhead than it is to improve the delta compression methods that were studied.

The XDFS results are good. Although it performs slightly slower than RCS at the data set with only a few versions per file, this is largely accounted for by the difference in initial insertion time, which includes the time required to create a new archive. XDFS is significantly faster than RCS with a sufficient number of versions per file due to better asymptotic performance. Disks are becoming cheaper and larger at a fast pace; the advantage in speed seems most desirable given current technology. The XDFS-f compression method, using Burns and Long's version jumping scheme, was the fastest method tested. It stores roughly twice as much data as its competitors, but retrieves versions using the minimal number of reads.

I have described this system as a platform for supporting version control systems and delta-compressed transport. Using this system, it is possible to build a delta compression-enabled application with minimal effort. XDFS accomplishes this with the use of an application-level file system layer, transaction support, a time–space tradeoff, and a critical feature for supporting delta-compressed transport.

Acknowledgements

I am grateful to Paul Hilfinger and Eric Brewer for their guidance and feedback. I would like to thank Randal Burns, Jeffrey Mogul, Andy Begel, and John Polstra for their comments on drafts of this paper. I would like to thank Sleepycat, Inc. for their assistance in working with the Berkeley Database.

Availability

XDFS and Xdelta are available under an open source license at http://xdelta.org.

References

[1] ALDERSON, A. A space-efficient technique for recording versions of data. *Software Engineering Journal 3*, 6 (Nov. 1988), 240–246.

[2] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), ACM SIGOPS, pp. 198–212.

[3] BERLINER, B. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA* (Berkeley, CA, USA, Jan. 1990), USENIX Association, Ed., USENIX, pp. 341–352.

[4] BLOCH, J. J. The camelot library: A C language extension for programming a general purpose distributed transaction system. In *9th International Conference on Distributed Computing Systems* (Washington, D.C., USA, June 1989), IEEE Computer Society Press, pp. 172–180.

[5] BURNS, R. C. Differential compression: A generalized solution for binary files. Master's thesis, University of California at Santa Cruz, Department of Computer Science, 1997.

[6] BURNS, R. C., AND LONG, D. D. E. Efficient distributed backup with delta compression. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems* (November 1997).

[7] CAREY, M., DEWITT, D., RICHARDSON, J., AND SHEKITA, E. Object and file management in the Exodus extensible database. *Proceedings of the Twelfth International Conference on Very Large Databses* (Aug. 1986).

[8] CHAN, M. C., AND WOO, T. Cache-based compaction: A new technique for optimizing Web transfer. In *Proceedings of the INFOCOM '99 conference* (Mar. 1999).

[9] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. The episode file system. In *Proceedings of the Usenix Winter 1992 Technical Conference* (Berkeley, CA, USA, Jan. 1991), Usenix Association, pp. 43–60.

[10] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to algorithms*, 6th ed. MIT Press and McGraw-Hill Book Company, 1992.

[11] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Principles* (New York, NY, USA, Dec. 1993), B. Liskov, Ed., ACM Press, pp. 15–28.

[12] DELCO, M., AND IONESCU, M. xProxy: A transparent caching and delta transfer system for web objects. http://www.cs.pdx.edu/~delco/xproxy.ps.gz, May 2000. UC Berkeley class project: CS262B/CS268.

[13] DEUTSCH, L. P., AND GAILLY, J.-L. RFC 1950: ZLIB compressed data format specification version 3.3, May 1996. Status: INFORMATIONAL.

[14] FRASER, C. W., AND MYERS, E. W. An editor for revision control. *ACM Transactions on Programming Languages and Systems 9*, 2 (Apr. 1987), 277–295.

[15] GAILLY, J. Personal communication. Comments in zlib source code file deflate.c, Feb. 1997.

[16] GANGER, G. R., AND KAASHOEK, M. F. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA* (Berkeley, CA, USA, Jan. 1997), USENIX, Ed., USENIX, pp. 1–17.

[17] GIAMPAOLO, D. *Practical file system design with the Be file system*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999. Includes comparison with Apple Macintosh, Linux, and Microsoft Windows file systems.

[18] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND JR, J. W. O. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Oct. 1991), ACM SIGOPS, pp. 16–25.

[19] GOPAL, B., AND MANBER, U. Integrating content-based access mechanism with hierarchical file systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, Feb. 199), USENIX Association, pp. 265–278.

[20] GRIMM, R., HSIEH, W. C., KAASHOEK, M. F., AND DE JONGE, W. Atomic recovery units: Failure atomicity for logical disks. In *ICDCS '96; Proceedings of the 16th International Conference on Distributed Computing Systems; May 27-30, 1996, Hong Kong* (Washington - Brussels - Tokyo, May 1996), IEEE, pp. 26–37.

[21] HAAS, L., CHANG, W., LOHMAN, G., MCPHERSON, M., WILMS, P., LAPIS, G., LINDSAY, B., PIRAHESH, H., CAREY, M., AND SHEKITA, E. Starburst mid-flight: As the dust clears. *tkde 2*, 1 (Mar. 1990), 143–160.

[22] HAERTEL, M., HAYES, D., STALLMAN, R., TOWER, L., AND EGGERT, P. Program source for GNU diffutils version 2.7. ftp://ftp.gnu.org/pub/gnu/diffutils/diffutils-2.7.tar.gz, 1998.

[23] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery management in QuickSilver. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 82–108.

[24] HEIDEMANN, J. S., AND POPEK, G. J. File-system development with stackable layers. *ACM Transactions on Computer Systems 12*, 1 (Feb. 1994), 58–89.

[25] HUNT, J. J., VO, K., AND TICHY, W. F. Delta algorithms: An empirical analysis. *ACMTSEM: ACM Transactions on Software Engineering and Methodology 7* (1998).

[26] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)* (New York, Oct. 5–8 1997), vol. 31,5 of *Operating Systems Review*, ACM Press, pp. 52–65.

[27] KORN, D., AND VO, K. IETF work-in-progress draft-korn-vcdiff-01.txt: The VCDIFF generic differencing and compression data format, Mar. 2000.

[28] LOPRESTI, D., AND TOMKINS, A. Block edit models for approximate string matching. *Theoretical Computer Science 181*, 1 (15 July 1997), 159–179.

[29] MACDONALD, J., HILFINGER, P. N., AND SEMENZATO, L. Prcs: The project revision control system. *Lecture Notes in Computer Science 1439* (July 1998), 33–45.

[30] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181–197.

[31] MILLER, W., AND MYERS, E. W. A file comparison program. *Software—Practice and Experience 15*, 11 (Nov. 1985), 1025–1040.

[32] MOGUL, J., KRISHNAMURTHY, B., DOUGLIS, F., FELDMANN, A., GOLAND, Y., AND VAN HOFF, A. IETF work-in-progress draft-mogul-http-delta-03.txt: Delta encoding in HTTP, Mar. 2000.

[33] MOGUL, J., AND VAN HOFF, A. IETF work-in-progress draft-mogul-http-digest-02.txt: Instance digests in HTTP, Mar. 2000.

[34] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference : Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM-97)* (New York, Sept.14–18 1997), vol. 27,4 of *Computer Communication Review*, ACM Press, pp. 181–196.

[35] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (Mar. 1992), 94–162.

[36] OLSON, M. A. The design and implementation of the inversion file system. Technical Report S2K-93-28, University of California, Berkeley, Apr. 1993.

[37] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference, FREENIX Track* (June 1999), pp. 183–192.

[38] POLSTRA, J. Program source for CVSup. ftp://ftp.freebsd.org/pub/FreeBSD/development/CVSup/, 1996.

[39] ROCHKIND, M. J. The source code control system. *IEEE Transactions on Software Engineering 1*, 4 (Dec. 1975), 364–370.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[41] ROSENTHAL, D. S. H. Requirements for a "stacking" vnode/vfs interface. Tech. Rep. SD-01-02-N014, UNIX International, Feb. 1992.

[42] SECHREST, S., AND MCCLENNEN, M. Blending hierarchical and attribute-based file naming. In *12th International Conference on Distributed Computing Systems* (Washington, D.C., USA, June 1992), IEEE Computer Society Press, pp. 572–580.

[43] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Usenix Annual Technical Conference* (1995).

[44] SELTZER, M. I. File system performance and transaction support. Technical Report ERL-93-1, University of California, Berkeley, Jan. 7, 1993.

[45] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (July 1981), 412–418. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[46] STONEBRAKER, M. The design of the POSTGRES storage system. In *vldb* (Brighton, England, Sept. 1987), P. Hammersley, Ed., pp. 289–300.

[47] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Usenix Annual Technical Conference* (Berkeley, CA, USA, Jan. 1996), USENIX Association, Ed., USENIX Conference Proceedings 1996, USENIX, pp. 1–14.

[48] THE FREEBSD PROJECT. Freebsd 3.2, USENIX edition. CDROM, June 1999.

[49] TICHY, W. F. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), 309–321.

[50] TICHY, W. F. RCS: A system for version control. *Software—Practice and Experience 15*, 7 (July 1985), 637–654.

[51] TRIDGELL, A., AND BARKER, P. rsync in http. http://linuxcare.com.au/rproxy/-calu_paper_html/index.html, July 1999.

[52] TRIDGELL, A., AND MACKERRAS, P. The Rsync algorithm. Tech. Rep. TR-CS-96-05, The Australian National University, June 1996.

[53] VAN HOFF, A., GIANNANDREA, J., HAPNER, M., CARTER, S., AND MEDIN, M. The http distribution and replication protocol. http://www.w3.org, Aug. 1997.

[54] VOGELS, W. File system usage in Windows NT 4.0. In *Proceedings of 17th ACM Symposium on Operating Systems Principles* (Dec. 1999), ACM SIGOPS, pp. 93–103.

[55] YU, L., AND ROSENKRANTZ, D. J. A linear-time scheme for version reconstruction. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 775–797.