Efficient Distributed Backup with Delta Compression

Randal C. Burns
Department of Computer Science
IBM Almaden Research Center
randal@almaden.ibm.com

Darrell D. E. Long†
Department of Computer Science
University of California, Santa Cruz
darrell@cs.ucsc.edu

Abstract

Inexpensive storage and more powerful processors have resulted in a proliferation of data that needs to be reliably backed up. Network resource limitations make it increasingly difficult to backup a distributed file system on a nightly or even weekly basis. By using delta compression algorithms, which minimally encode a version of a file using only the bytes that have changed, a backup system can compress the data sent to a server. With the delta backup technique, we can achieve significant savings in network transmission time over previous techniques.

Our measurements indicate that file system data may, on average, be compressed to within 10% of its original size with this method and that approximately 45% of all changed files have also been backed up in the previous week. Based on our measurements, we conclude that a small file store on the client that contains copies of previously backed up files can be used to retain versions in order to generate delta files.

To reduce the load on the backup server, we implement a modified version storage architecture, *version jumping*, that allows us to restore delta encoded file versions with at most two accesses to tertiary storage. This minimizes server workload and network transmission time on file restore.

1 Introduction

Currently, file system backup takes too long and has a prohibitive storage cost. Resource constraints impede the reliable and frequent backup of the increasing amounts of data spinning on disks today. The time required to perform backup is in direct proportion to the amount of data transmitted over the network from backup clients to a backup server. By using *delta compression*, compactly encoding a file version as a set of changes from a previous version, our backup system architecture reduces the size of data to be transmitted, reducing both time to perform backup and storage required on a backup server.

Backup and restore can be limited by both network bandwidth, often 10 Mb/s, and poor throughput to tertiary storage devices, as slow as 500 KB/s to tape storage. Since resource limitations frequently make backing up changed files infeasible over a single night or even weekend, delta file compression has great potential to alleviate bandwidth problems by using available client processor cycles and disk storage to reduce the amount of data transferred. This technology enables us to perform file backup over lower bandwidth channels than were previously possible, for example a subscription based backup service over the Internet.

Early efforts to reduce the amount of data to be backed up produced a simple optimization, *incremental backup*, which backs up only those files that have been modified since the end of the last *epoch*, the period between two backups. While incremental backup only detects changes at the granularity of a file, *delta backup* refines this concept, transmitting only the altered bytes in the files to be incrementally backed up [12]. Consequently, if only a few bytes are modified in a large file, the backup system saves the expense of transmitting the large file in favor of transmitting only the changed bytes.

Recent advances in differencing algorithms [1, 4], allow nearly optimally compressed encodings of binary inputs in linear time. We use such an algorithm to generate delta encodings of versions.

A differencing algorithm finds and outputs the changes made between two versions of the same file by locating common strings to be copied and unique strings to be added explicitly. A *delta file* (Δ) is the encoding of the output of a differencing algorithm. An algorithm that creates a delta file takes as input two versions of a file, a *reference file* and a *version file* to be encoded, and outputs a delta file representing the modifications made between versions:

† The work of this author was performed while a Visiting Scientist at the IBM Almaden Research Center.

$$V_{\text{reference}} + V_{\text{version}} \rightarrow \Delta_{(V_{\text{reference}}, V_{\text{version}})}.$$

Reconstruction, the inverse operation, requires the reference file and a delta file to rebuild a version:

$$V_{\text{reference}} + \Delta_{(V_{\text{reference}}, V_{\text{version}})} \rightarrow V_{\text{version}}.$$

A backup system can leverage delta files to generate minimal file representations (see Figure 1). We enhanced the client/server architecture of the AdStar Distributed Storage Manager (ADSM) backup system [5] to transmit delta files when a backup client has retained two versions of the same file. Furthermore, both uncompressed files and delta encoded files still realize benefits from the standard file compression methods that ADSM already utilizes [1]. We integrate delta backup into ADSM and have a backwardly compatible system with optimizations to transmit and store a reduced amount of data at the server.

The server storage and network transmission benefits are paid for in the client processor cycles to generate delta files and in additional disk storage at a backup client to retain second copies of files that are used to generate delta files. This architecture optimizes the network and storage bottleneck at the server in favor of the distributed resources of a server's multiple clients.

We will describe previous work in the use of delta compression for version storage in §2. Our modifications to the ADSM system architecture are presented in §3. In §4, we describe the delta storage problem in the presence of a large number of versions. In §5, we analyze the performance of the version jumping storage method and compare it to the optimally compact storage method of linear delta chains. In §6 potential future work is discussed and we present our conclusions in §7.

2 Origins of Delta Backup

Delta backup emerged from many applications, the first instance appearing in database technology. The database pages that are written between backup epochs are marked as "dirty" using a single bit [10, 17]. At the end of an epoch, only the dirty pages need to be backed up. This concept parallels delta backup but operates only at page granularity. For file systems, there are no guarantees that modifications have page alignment. While dirty bits are effective for databases, they may not apply well to file system backup.

To improve on the granularity of backup, logging methods have been used to record the changes to a database [9, 14] and a file system [16]. A logging system records every write during an epoch to a log file. This can be used to recover the version as it existed at the start of an epoch into its current state. While semantically similar to delta compression, logging does not provide the compressibility guarantees of differencing algorithms. In the database example, a log or journal of changes grows in proportion to the number of writes made. If the same bytes are modified more than once or if adjacent bytes are modified at different times, this will not be a minimal representation. For an extremely active page, the log will likely exceed the page size. Differential compression also has the guarantee that a log of modifications to a file will be no smaller than the corresponding delta [4].

Recently, several commercial systems have appeared on the marketplace that claim to perform delta backup [11, 6, 18]. While the exact methods these systems use have not been disclosed, the product literature implies that they either perform logging [11] or difference at the granularity of a file block [18, 6]. We perform delta backup at the granularity of a byte. By running differential compression algorithms on the changed versions at this granularity, we generate and backup minimal delta files.

2.1 Previous Work in Version Management

Despite file restoration being the infrequent operation for backup and restore, optimizing this process has great utility. Often, restore is performed when file system components have been lost or damaged. Unavailable data and nonfunctional systems cost businesses, universities and home users lost time and revenue. This contrasts the backup operation which is generally performed at night or in other low usage periods.

Restoring files that have been stored using delta backup generates additional concerns in delta file management. Traditional methods for storing deltas require the decompression agent to examine either all of the versions of a given file [13] or all versions in between the first version and the version being restored [15]. In either case, the time to reconstruct a given file grows at least linearly (see §4.1) with respect to the number of versions involved.

A backup system has the additional limitation that any given delta file may reside on a physically distinct media device and device access can be slow, generally several seconds to load a tape in a tape robot [8]. Consequently, having many distinct deltas participate in the restoration of a single file becomes costly in device latency. An important goal of our system is to minimize the number of deltas participating in any given restore operation.

Previous efforts in the efficient restoration of file versions have provided restoration execution times independent of the number of intermediate versions. These include methods based on AVL Dags [7], linked data structures [19, 2], or string libraries [3]. However, these require all delta versions of a given file to be present at restore time and are consequently infeasible for a backup system. Such a backup system would require all prior versions of a file to be recalled from long term storage for that file to be reconstructed.

As previous methods in efficient restoration fail to ap-

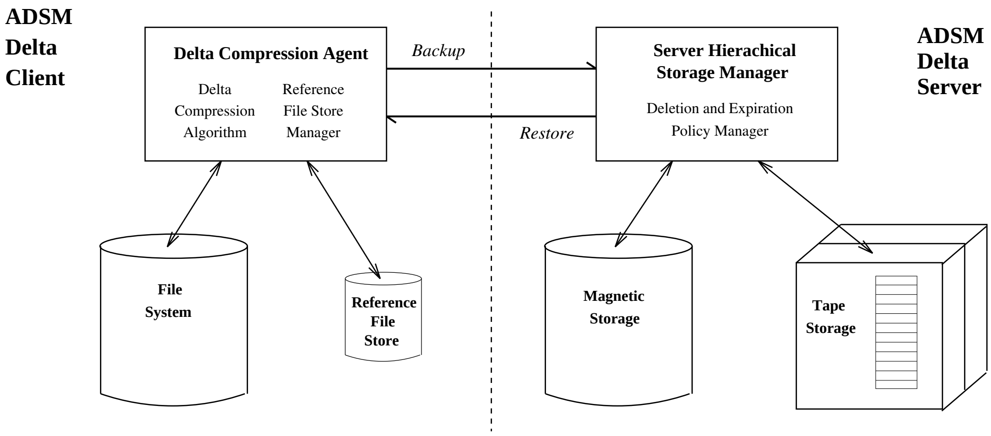


Figure 1: Client/server schematic for a delta backup system.

ply to the backup and restore application, we describe a new technique called *version jumping* and an associated system architecture. Version jumping takes many delta versions off of the same reference version and consequently requires at most two files to perform the restore operation on a given version. The restoration time is also independent of the total number of stored versions.

3 System Architecture

We modified the architecture and design of the AdStar Distributed Storage Manager (ADSM) from IBM to add delta compression capabilities. The modified ADSM client delta compresses file data at the granularity of a byte and transmits delta files to the server. The modified ADSM server has enhanced file deletion capabilities to ensure that each delta file stored on the server also has the corresponding reference file.

ADSM is a client/server backup and restore system currently available for over 30 client platforms. The ADSM server is available for several operating systems including Windows NT and various UNIX and mainframe operating systems. The key features of ADSM are scheduled policy management for file backup, both client request and server polling, and hierarchical storage management of server media devices including tape, disk drive and optical drives.

3.1 Delta Compression at the Client

We have modified the standard ADSM client to perform delta backup and restore operations. The modifications include the differencing and reconstruction algorithms (see §1) and the addition of a *reference file store.*

In order to compute deltas, the current version of the file must be compared with a reference version. We have the choice of storing the reference version on the client or fetching it from the server. Obtaining the reference file from the server is unviable since it would increase both network traffic and server load, adversely affecting the time to perform the backup. By storing the reference file at the client, we incur a small local storage cost in exchange for a large benefit in decreased backup time.

We considered several options for maintaining the reference files, including copy-on-write, shadowing, and file system logging. Each of these options had to be rejected since they violated the design criterion that no file system modifications could be made. Since ADSM supports more than 30 client operating systems, any file system modification presents significant portability and maintenance concerns. Instead, we chose to keep copies of recently backed up files in a reference file store.

The reference file store is a reserved area on disk where reference versions of files are kept (see Figure 1). When sending an uncompressed file to its server, the backup client copies this file to its reference file store. At this point, the file system holds two copies of the same file: one active version in file system space and a static reference version in the reference file store. When the reference file store fills, the client selects a file to be ejected. We choose the victim file with a simple weighted least recently used (LRU) technique. In a backup system, many files are equally old, as they have been backed up at the same epoch. In order to discern among these multiple potential victims, our backup client uses an additional metric to weight files of equal LRU value. We select as a victim the reference file that achieved the worst relative compression on its previous usage, *i.e* . the file with the highest delta file size to uncompressed file size ratio at the last backup. This allows us to discern from many potential victims to increase the utility of our reference file store. At the same time, this weighting does not violate the tried and true LRU principle and consequently can be expected to realize all of the benefits of locality.

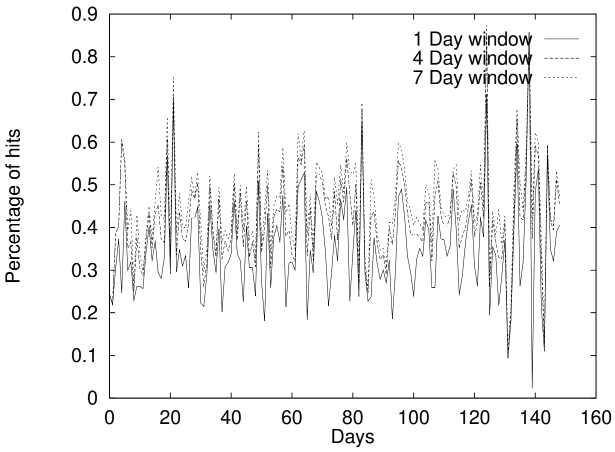


Figure 2: Fraction of files modified today also modified 1, 4 and 7 days ago.

When the backup client sends a file that has a prior version in the reference file store, the client delta compresses this file and transmits the delta version. The old version of the file is retained in the reference file store as a common reference file for a version jumping system.

We do not expect the storage requirements of the reference file store to be excessive. In order to evaluate the effectiveness of the reference file store, we collected data for five months from the file servers at the School of Engineering at the University of California, Santa Cruz. Each night, we would search for all files that had either been modified or newly created during the past 24 hours. Our measurements indicate that of approximately 2,356,400 files, less than 0.55% are newly created or modified on an average day and less than 2% on any given day. These recently modified files contain approximately 1% of the file system data.

In Figure 2, we estimate the effectiveness of the reference file store using a sliding window. We considered windows of 1–7 days. The x-axis indicates the day in the trace, while the y-axis denotes the fraction of files that were modified on that day and that were also created or modified in that window. Files that are not in the window are either newly created or have not been modified recently. An average of 45% and a maximum of 87% of files that are modified on a given day had also been modified in the last 7 days. This locality of file system access verifies the usefulness of a reference file store and we expect that a small fraction, approximately 5%, of local file system space will be adequate to maintain copies of recently modified file versions.

3.2 Delta Versions at the Server

In addition to our client modifications, delta version storage requires some enhanced file retention and file garbage collection policies at a backup server. For example, a naïve server might discard the reference files for delta files that can be recalled. The files that these deltas encode would then be lost from the backup storage pool. The file retention policies we will describe require additional metadata to resolve the dependencies between a delta file and its corresponding reference file.

A backup server accepts and retains files from backup clients (see Figure 1). These files are available for restoration by the clients that backed them up for the duration of their residence on the server. Files that are available for restoration are *active* files. A typical file retention policy would be: "hold the four most recent versions of a file." While file retention policies may be far more complex, this turns out to have no bearing on our analysis and consequent requirements for backup servers. We only concern ourselves with the existence of deletion policies on an ADSM server and that files on the server are only active for a finite number of backup epochs.

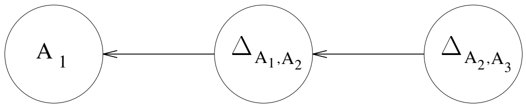
To reconstruct a version file to a client from a backup server, when the server has a delta encoding of this file, the client must restore both the delta encoding and the corresponding reference file. Under this constraint, a modified retention policy dictates that a backup server must retain all reference files for its active files that are delta encoded. In other words, a file cannot be deleted from the backup server until all active files that use it as a reference file have also been deleted. This relationship easily translates to a dependency digraph with dependencies represented by directed edges and files by nodes. This digraph is used both to encode dependent files which need to be retained and to garbage collect the reference files when the referring delta versions are deleted.

For a version chain storage scheme, we may store a delta file, Δ_{A_2,A_3}, which depends on file A_2. However, the backup server may not store A_2. Instead it stores a delta file representation. In this event, we have Δ_{A_2,A_3} depend upon the delta encoding of A_2, Δ_{A_1,A_2} (see Figure 3(a)).

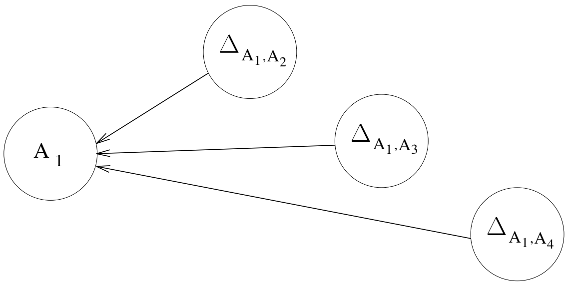
The dependency digraphs for delta chains never branch and do not have cycles. Furthermore, each node in the digraph has at most one inbound edge and one outbound edge. In this example, to restore file A_3, the backup server needs to keep its delta encoding, Δ_{A_2,A_3}, and it needs to be able to construct the file it depends upon, A_2. Since, we only have the delta encoding of A_2, we must retain this encoding, Δ_{A_1,A_2}, and all files it requires to reconstruct A_2. The obvious conclusion is: given a dependency digraph, to reconstruct the file at a node, the backup server must retain all files in nodes reachable from the node to be reconstructed.

For version jumping, as in version chains, the version digraphs do not branch. However, any node in the version digraph may have multiple incoming edges, *i.e.* a file may be a common reference file among many deltas (see Figure 3(b)).

The dependency digraphs for both version jumping and



(a) Simple delta chain.



(b) Jumping version chain.

Figure 3: Dependency graphs for garbage collection and server deletion policies.

delta chains are acyclic. This results directly from delta files being generated in a strict temporal order; the reference file for any delta file is always from a previous backup epoch. Since these digraphs are acyclic, we can implicitly solve the dependency problem with local information. We decide whether or not the node is available for deletion without traversing the dependency digraph. This method works for all acyclic digraphs.

At each node in the dependency digraph, *i.e.* for each file, we maintain a pointer to the reference file and a reference counter. The pointer to the reference file indicates a dependency; a delta encoded version points to its reference file and an uncompressed file has no value for this pointer. The reference counter stores the number of inbound edges to a node and is used for garbage collection. A node has no knowledge of what files depend on it, only that dependent files exist.

When backing up a delta file, we require a client to transmit its file identifier and the identifier of its reference file. When we store a new delta file at the ADSM server, we store the name of its reference file as its dependent reference and initialize its reference counter to zero. We must also update the metadata of the referenced file by incrementing its reference counter. With these two metadata fields, the modified backup server has enough information to retain dependent files and can guarantee that the reference files for its active files are available.

3.3 Two-phase deletion

When storing delta files, the backup server often needs to retain files that are no longer active. We modify deletion in this system to place files into three states: active files, inactive dependent files, and inactive independent files. Inactive dependent files are those files that can no longer be restored as a policy criterion for deletion has been met but need to be retained as active delta versions depend on them. Inactive independent files are those files suitable for removal from storage as they cannot be restored and no active delta files depend on them.

We add one more piece of metadata, an activity bit, to each file backed up on our server. When a file is received at the server, it is stored as usual and its activity bit is set. This file is now marked as *active*. The backup server implements deletion policies as usual. However, when a file retention policy indicates that a file can no longer be restored, we clear the file's activity bit instead of deleting it. Its state is now *inactive*. Clearing the activity bit is the first phase of deletion. If no other files depend on this newly inactive file, it may be removed from storage, the second phase.

Marking a file inactive may result in other files becoming independent as well. The newly independent reference file is garbage collected through the reference pointer of the delta file. The following rules govern file deletion:

- When a file has no referring deltas, *i.e.* its reference counter equals zero, delete the file.
- When deleting a delta file, decrement the reference counter of its reference file and garbage collect the reference file if appropriate.

Reference counting, reference file pointers, and activity bits correctly implement the reachability dependence relationship. The two phase deletion technique operates locally, never traversing the implicit dependency digraph, and consequently incurs little execution time overhead. The backup server only traverses this digraph when restoring files. It follows dependency pointers to determine the set of files required to restore the delta version in question.

4 Delta Storage and Transmission

The time to transmit files to and from a server is directly proportional to the amount of data sent. For a delta backup and restore system, the amount of data is also related to the manner in which delta files are generated. We develop an analysis to compare the version jumping storage method with storing delta files as version to version incremental changes. We show that version jumping pays a small compression penalty for file system backup when compared to the optimal linear delta chains. In exchange for this lost compression, version jumping allows a delta file to be rebuilt with at most two accesses to tertiary storage.

The version storage problem for backup and restore differs due to special storage requirements and the distributed nature of the application. Since ADSM stores files on multiple and potentially slow media devices, not all versions of a file are readily available. This unavailability and other design considerations shape our methods for storing delta versions. If a backup system stores files on a tape robot then, when reconstructing a file, the server may have to load a separate tape for every delta it must access. Access to each tape may require several seconds. Consequently, multiple tape motions are intolerable. Our version jumping design guarantees that at most two accesses to the backing store are needed to rebuild a delta file.

Also, we minimize the server load by performing all of the differencing tasks on the client. Since a server processes requests from many clients, a system that attempted to run compression algorithms at this location would quickly become processor bound and achieve poor data throughput. For client side differencing, we generate delta files at the client and transmit them to the server where they are stored unaltered. Using client differencing, the server incurs no additional processor overhead.

4.1 Linear Delta Chains

Serializability and lack of concurrency in file systems result in each file having a single preceeding and single following version. This linear sequence of versions forms a history of modifications to a file which we call a *version chain*. We now develop a notation for delta storage and analyze a linear version chain stored using traditional methods [15].

Linear delta chains are the most compact version storage scheme as the inter-version modification are smallest when differencing between consecutive versions. We will use the optimality of linear delta chains later as a basis to compare the compression of the version jumping scheme.

We denote the uncompressed i^{th} version of a file by V_i. The difference between two versions V_i and V_j is indicated by $\Delta_{(V_i,V_j)}$. The file $\Delta_{(V_i,V_j)}$ should be considered the differentially compressed encoding of V_j with respect to V_i such that V_j can be restored by the inverse differencing operation applied to V_i and $\Delta_{(V_i,V_j)}$. We indicate the differencing operation by

$$\delta(V_i, V_j) \rightarrow \Delta_{(V_i,V_j)}$$

and the inverse differencing or reconstruction operation by

$$\delta^{-1}(\Delta_{(V_i,V_j)}, V_i) \rightarrow V_j.$$

By convention, V_i is created by modification of V_{i-1}. For versions V_i and V_j in a linear version chain, these versions are adjacent if they obey the property $j - i = 1$ and an intermediate version V_k obeys the property $i < k < j$.

For our analysis, we consider a linear sequence of versions of the same file that continues indefinitely,

$$V_1, V_2, \ldots, V_{i-1}, V_i, V_{i+1}, \ldots.$$

The traditional way to store this version chain as a series of deltas is, for two adjacent versions V_i and V_{i+1}, to store the difference between these two files, $\Delta_{V_i,V_{i+1}}$ [13]. This produces the following "delta chain"

$$V_1, \Delta_{(V_1,V_2)}, \ldots, \Delta_{(V_{i-1},V_i)}, \Delta_{(V_i,V_{i+1})}, \ldots.$$

Under this system, to reconstruct an arbitrary version V_i, the algorithm must apply the inverse differencing algorithm recursively for all intermediate versions 2 through i. This relation can be compactly expressed as a recurrence. V_i represents the contents of the i^{th} version of a file and R_i is the recurrent file version. So when rebuilding V_i, $V_i = R_i$ and

$$R_i = \delta^{-1}(\Delta_{(V_{i-1},V_i)}, R_{i-1}); \quad R_1 = V_1.$$

The time required to restore a version depends upon the time to restore all of the intermediate versions. In general, restoration time grows linearly in the number of intermediate versions. In a system that retains multiple versions, the cost of restoring the most remote version quickly becomes exorbitant.

4.2 Reverse Delta Chains

Some version control systems solve the problem of long delta chains with reverse delta chain storage [15]. A reverse delta chain keeps the most recent version of a file present and uncompressed. The version chain is then stored as a set of backward deltas. For most applications, the most recent versions are accessed far more frequently than older versions and the cost of restoring an old version with many intermediate versions is offset by the low probability of that version being requested.

We have seen that linear delta chains fail to provide acceptable performance for delta storage (see §4.1). Our design constraints of client-side differencing and two tape accesses for delta restore also eliminate the use of reverse delta chains. We show this by examining the steps taken in a reverse delta system to transmit the next version to the backup server.

At some point, a server stores a reverse delta chain of the form

$$\Delta_{(V_2,V_1)}, \Delta_{(V_3,V_2)}, \ldots, \Delta_{(V_n,V_{n-1})}, V_n.$$

In order to backup its new version of the file, V_{n+1}, the client generates a difference file, $\Delta_{(V_n,V_{n+1})}$ and transmits this difference to the server. However, this delta is not the file that the server needs to store. It needs to generate and store

$\Delta_{(V_{n+1},V_n)}$. Upon receiving $\Delta_{(V_n,V_{n+1})}$, the server must apply the difference to V_n to create V_{n+1} and then run the differencing algorithm to create $\Delta_{(V_{n+1},V_n)}$. To update a single version in the reverse delta chain, the server must store two new files, recall one old file, and perform both the differencing and reconstruction operations. Reverse delta chains fail to meet our design criteria as they implement neither minimal server processor load nor reconstruction with the minimum number of participating files.

4.3 Version Jumping Delta Chains

Our solution to the version storage implements what we call *jumping deltas*. This design uses a minimum number of files for reconstruction and performs differencing on the backup client.

In a version jumping system, the server stores versions in a modified forward delta chain with an occasional whole file rather than a delta. Such a chain looks like:

$$V_1, \Delta_{(V_1,V_2)}, \Delta_{(V_1,V_3)}, \ldots, \Delta_{(V_1,V_{i-1})}, V_i, \Delta_{(V_i,V_{i+1})}, \ldots$$

Storing this sequence of files allows any given version to be reconstructed by accessing at most two files from the version chain.

When performing delta compression at the backup client, the files transmitted to the server may be stored directly without additional manipulation. This *immediate storage* at the server limits the processor overhead associated with each client session and optimizes the backup server.

An obvious concern with these methods is that one expects compression to degrade when taking the difference between two non-adjacent versions, *i.e.* for versions V_i and V_j, $|\Delta_{(V_i,V_j)}|$[1] increases as $j-i$ increases. Since the compression is likely to degrade as the version distance, $j-i$, increases, we require an occasional whole file to limit the maximum version distance. This raises the question: what is the optimal number of versions between whole files?

5 Performance Analysis

We analyze the storage and transmission cost of backing up files using a version jumping policy. We already know that version jumping far outperforms other version storage methods on restore, since it requires only two files to be accessed from tertiary storage to restore a delta file. Now, by showing that the compression loss with version jumping is small as compared to linear delta chains, the optimal storage method for delta compression, we conclude version jumping to be a superior policy.

The analysis of transmission time and server storage is identical, since our backup server immediately stores all files, including deltas, that it receives and transmission time is in direct proportion to the amount of data sent. We choose to examine storage for ease of understanding the contents of the backup server at any given time and use this analysis to draw conclusions about transmission times as well.

5.1 Version Jumping Chains

Consider a set of versions, $V_1, \ldots, V_i, \ldots$, where any two adjacent versions, V_i and V_{i+1}, have $\alpha|V_i|$ modified symbols between them. The parameter α represents the *compressibility* between adjacent versions. An ideal differencing algorithm can create a delta file, $\Delta_{(V_i,V_{i+1})}$, with maximum size $\alpha|V_i|$. The symbols encoded in a delta file can either replace existing symbols or add data to the file, as all reasonable encodings do not mark deleted symbols [1]. The compression achieved on version V_i is given by

$$1 - \frac{|\Delta_{(V_i,V_{i+1})}|}{|V_{i+1}|}.$$

Since we are considering the relative compressibility of all new versions with the same size deltas, the delta file can be as large as size $\alpha|V_i|$ and the new version ranges in size from $|V_i|$ to $(1+\alpha)|V_i|$. Consequently, the worst case compression occurs when the $\alpha|V_i|$ modified symbols in V_{i+1} replace existing symbols in V_i, *i.e.* $|V_i| = |V_{i+1}|$. The worst case occurs when the file stays the same size.

Between versions V_i and V_{i+1}, there are a maximum of $\alpha|V_i|$ modified symbols and between versions V_{i+1} and V_{i+2} there are at most $\alpha|V_{i+1}|$ modified symbols. By invoking the union bound on the number of modified symbols between versions V_i and V_{i+2}, there are at most $2\alpha|V_i|$ modified symbols, assuming worst case compression. This occurs when the changed symbols between versions are disjoint and the versions are the same size. Generalizing this argument to n intermediate versions, we can express the worst case size of the jumping delta between V_1 and V_n as $n\alpha|V_1|$. Having defined the size of an arbitrary delta, we can determine how much storage is required to store a linear set of n versions using the jumping delta technique

$$S(n) = |V_1| + \sum_{i=2}^{n} |\Delta_{(1,i)}| \leq |V_1| \left(1 + \alpha \sum_{i=2}^{n} i\right). \qquad (1)$$

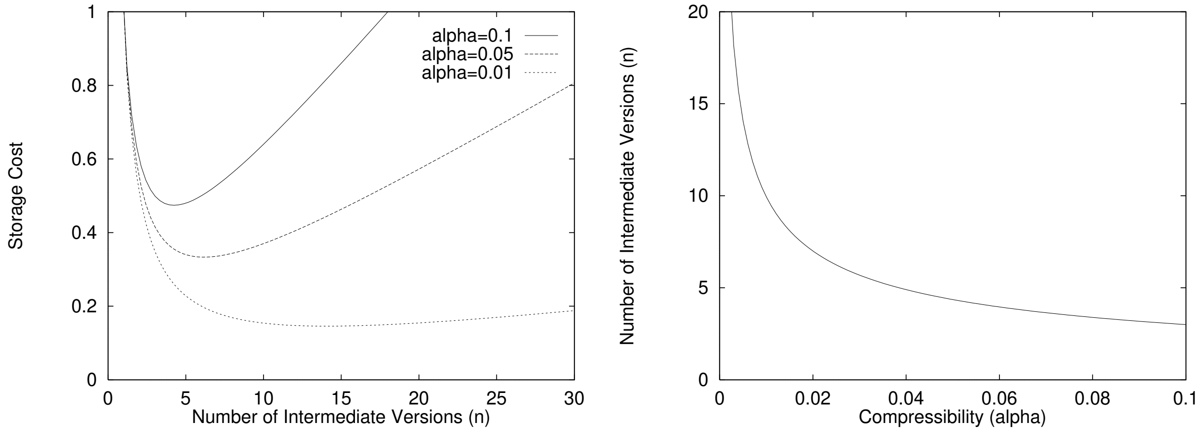
We are also interested in determining the optimal number of jumping deltas to be taken between whole files. We do this by minimizing the average cost of storing a version as a function of n, the number of versions between whole files. The average cost of storing an arbitrary version is

$$s(n) = \frac{S(n)}{n} \leq \frac{|V_1|}{2n} \left(\alpha n^2 + \alpha n - 2\alpha + 2\right). \qquad (2)$$

This function has a minimum with respect to n at

$$n = \frac{\sqrt{2\alpha(1-\alpha)}}{\alpha}. \qquad (3)$$

[1]For a file V, we use $|V|$ to denote the size of the file. Since files are one dimensional streams of bytes, this is synonymous to the length of V.



(a) Average Storage Cost in units of $|V_1|$.

(b) Number of intermediate versions at minimum of $s(n)$.

Figure 4: The per version transmission and storage cost in the worst case parameterized by compression (α).

Equation 2 expresses the worst case per version storage cost, and consequently the per version transmission cost, of keeping delta files using version jumping. For any given value of α, the optimal number of jumping deltas between uncompressed versions is given by the minimum of Equation 2. We give a closed form solution to this minimum in Equation 3.

Figure 4 displays both the version storage cost parameterized by α and the minimum of this family of curves as a function of α. We see that for large values of n, version jumping provides poor compression, as the version distance increases and the compression degrades as expected. However, there is an optimal number of versions, depending upon compressibility, at which version jumping performs reasonably. When the number of transmitted versions exceeds the number at which the storage cost is minimum (see Equation 3), the system's best decision is to transmit the next file without delta encoding and start a new version jumping sequence.

Our analysis is the worst case compression bound. In practice, we expect to achieve much better compression, as the version to version changes will not be completely disjoint and files will both grow and shrink in size.

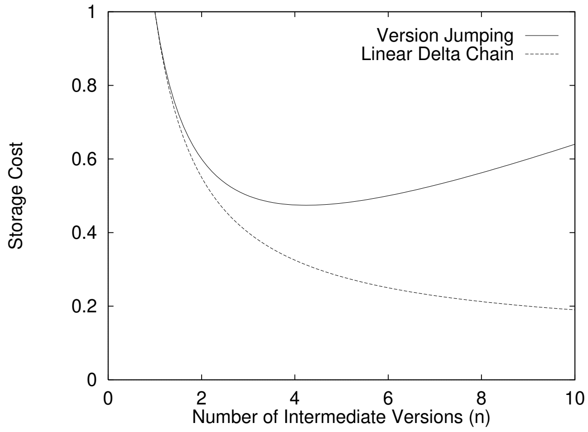
Also, we cannot expect to detect the minimum of the storage and transmission curve analytically, since α will not be constant. Instead, a backup client that implements version jumping monitors the size of the delta files as compared to the corresponding uncompressed files. When the average compression, total bytes in transmitted files over total bytes in uncompressed files, stops decreasing, compression is degraded past a minimum, similar to the curve in Equation 2. At this point the client transmits a new uncompressed file to start a new jumping version chain. This minimum could be local, as this policy is only a heuristic for detecting the minimum. However, in general, files differ more as the version distance increases and the heuristic will detect the global minimum.

5.2 Linear Delta Chains

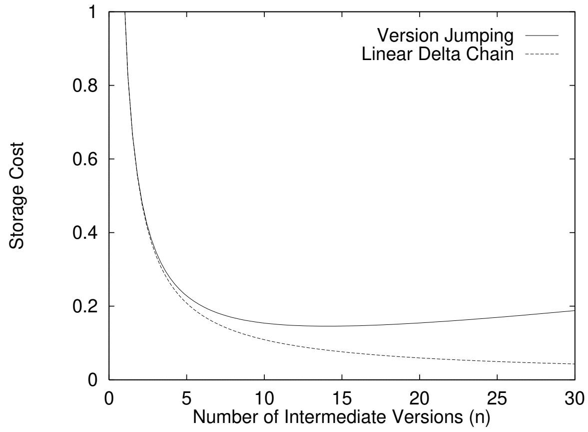
Having developed an expression for the worst case per version storage cost for version jumping (see §5.1), we do the same for linear delta chains. Recall that linear delta chains are not suitable for backup and restore (see §4.1) but they do provide a bound on the best possible compression performance of a a version storage architecture. Version jumping provides constant time reconstruction of any version. To realize this efficient file restore, we trade a small fraction of potential compression. We quantify this loss of compression with respect to the optimally compressing linear delta chains.

We bound compression degradation by deriving an expression for the per version storage cost under a linear delta chain and comparing this to Equation 2. The limited loss in compression for version jumping is offset by decreased restore time and we conclude that version jumping is the superior policy for backup and restore.

Several facts about the nature of delta storage for backup and restore apply to our analysis. First, a backup and restore storage system must always retain at least one whole file in order to be able to reconstruct versions. Additionally, a backing store holds a bounded number of file system backups. We let the number of backup versions retained be given by the parameter n and can then say that, for any file, a backing store must retain at least one uncompressed version of that file and at most $n - 1$ deltas based on that uncompressed version.



(a) α=0.1



(b) α=0.01

Figure 5: The relative per version storage and transmission cost, in units of $|V_1|$, of version jumping, Equation 2, compared to delta chaining, Equation 5.

We derive an expression for the amount of storage used by a linear delta chain. The minimal delta chain on n files contains some reference file of size $|V_1|$ and $n-1$ delta files of size $\alpha|V_j|$ for all j between 1 and $n-1$. Using the same assumptions about version to version modifications that were used in Section 5.1, the total storage required for an n version linear delta chain is given by:

$$C(n) = |V_1| + \sum_{i=2}^{n} |\Delta_{(V_{i-1},V_i)}| \leq |V_1| \left(1 + (n-1)\alpha\right). \tag{4}$$

and the average storage required for each version is:

$$c(n) = \frac{C(n)}{n} \leq \frac{|V_1|}{n} \left(1 + (n-1)\alpha\right). \tag{5}$$

In Figure 5, we compare the relative per file cost of storing versions in a linear delta chain with the per version storage cost of version jumping. Based on experimental results [4], we chose $\alpha = 0.01$ and $\alpha = 0.1$ as a low and a high value for the compressibility of file system data. We note that the version jumping and delta chain storage curves are nearly identical for small values of n. For large values of n, the compression of version jumping degrades and the curves diverge. However, at these larger values of n, the restore time with delta chains grows linearly larger with the number of versions (see §4.1). In addition to the asymptotic growth of the restore function, linear chains also require multiple accesses to slow media devices, which compounds the restore problem. As the number of intermediate versions stored grows, the restore cost quickly renders linear version chain storage intolerable.

For both delta chains and version jumping, the number of intermediate versions will need to be kept small. In version jumping it is desirable to pick the largest value of n less than or equal to the minimum value of the storage function (see Equation 3). For linear delta chains n must be kept small to make file restore times reasonable as restore time grows in the size of the versions and retrieving these files generally requires access to slow tape devices. At these small values of n, version jumping is a superior policy as it compresses nearly as well and requires two tape accesses to restore a file.

Fortunately, backup and restore applications generally require few versions of a file to be stored at any given time. An organization that retains daily backups for a week and weekly backups for a month would be considered to have a very aggressive backup policy. This configuration would make n valued at 9. For the majority of configurations, n will take on a value between 2 and 10. While some applications may exist that require more versions, the expense of storage and storage management combined with data becoming older and consequently less pertinent tends to limit the number of versions kept in a backup system.

An operational jumping delta backup system will perform much better than this worst case analysis as many of the worst case factors are not realized on file system data. In particular, the modifications from version to version will not be completely disjoint and versions of a file should change in size. Consequently, we conclude that our system can maintain more deltas between whole files than this analysis specifies. Worst case analysis does allow us to assert the viability of a version jumping system. As the worst case bounds are plausible for the application, a delta backup system improves on these bounds providing a viable backup architecture.

6 Future Work

While maintaining a reference file store allows recent version modifications to be stored in a small fraction of the file system space, large files presents a concern as they may consume significant storage space in the reference file store. We believe that there is merit to considering block based reference file storage schemes, combined block and file storing, and finally, using digital signatures to compactly "copy" a representation of large files and files that have been ejected from the reference file store.

The reference store could choose to copy blocks rather than files. This would allow only the modified blocks in a changed file to be duplicated in the reference store. While this may mitigate the large file problem, it prevents a differencing algorithm from detecting changes in multi-block files that are not block aligned. The reference store could instead choose to save whole files for most files and only store blocks for large files. Such a combined scheme could heuristically address both the large file and block alignment issues. Finally, to save storage on large files, the file blocks could be uniquely identified using digital signatures. This greatly reduces the storage cost but only permits delta files to be calculated at a block granularity.

Our version jumping technique allows delta files to be restored with two accesses to the backup server storage pool. Generally, this means that two tapes must be loaded, each requiring several seconds. However, a backup server that could collocate delta files and reference files on the same tape could access both files by loading a single tape. Collocation of delta files would provide a significant performance gain for file restore but would require extra tape motions when files are backed up or migrated from a different storage location.

7 Conclusions

By using delta file compression, we modified ADSM to send compact encodings of versioned data reducing both the network transmission time and the server storage cost. We have presented an architecture based on the version jumping method for storing delta files at a backup server, where many delta files are generated from a common reference file. We have shown that version jumping far outperforms previous methods for file system restore, as it requires only two accesses to the server store to rebuild delta files. At the same time, version jumping pays only small compression penalties when generating delta files for file system backup.

Previous methods for efficient restore were examined and determined to not fit the problems requirements as they require all delta files to be available simultaneously. Methods based on delta chains may require as many accesses to the backing store as there are versions on the backup server. As any given file may reside on physically distinct media, and access to these devices may be slow, previous methods failed to meet the special needs of delta backup. We then conclude that version jumping is a practical and efficient way to limit restore time by making small sacrifices in compression.

Modifications to both the backup client and server help support delta backup. We described a system where the client maintains a store of reference files so that delta files may be generated for transmission and storage. We have also described enhanced file deletion and garbage collection policies at the backup server. The server determines which files are dependent, those inactive files that must be retained in order to reconstruct active delta files.

Acknowledgments

We are indebted to J. Gray of Microsoft Research for his review of our work and aid in shepherding this paper into its final form. Many thanks to L. Stockmeyer and M. Ajtai of the IBM Almaden Research Center whose input helped shape the architecture we present. We would also like to extend our thanks to L. You who helped in the design and implementation of this system and to N. Pass, J. Menon, and R. Golding whose support and guidance are instrumental to the continuing success of our research.

References

[1] Miklos Ajtai, Randal Burns, Ronald Fagin, and Larry Stockmeyer. Efficient differential compression of binary sources. IBM Research: *In Preparation*, 1997.

[2] Albert Alderson. A space-efficient technique for recording versions of data. *Software Engineering Journal*, 3(6):240–246, June 1988.

[3] Andrew P. Black and Charles H. Burris, Jr. A compact representation for file versions: A preliminary report. In *Proceedings of the 5th International Conference on Data Engineering*, pages 321–329. IEEE, 1989.

[4] Randal Burns and Darrell D. E. Long. A linear time, constant space differencing algorithm. In *Proceedings of the 1997 International Performance, Computing and Communications Conference (IPCCC'97), Feb. 5-7, Tempe/Phoenix, Arizona, USA*, February 1997.

[5] Luis-Felipe Cabrera, Robert Rees, Stefan Steiner, Michael Penner, and Wayne Hineman. ADSM: A multi-platform, scalable, backup and archive mass storage system. In *IEEE Compcon, San Francisco, CA March 5-9, 1995*, pages 420–427. IEEE, 1995.

[6] Connected Corp. *The Importance of Backup in Small Business.* http://www.connected.com/wtpaper.html, 1996.

[7] Christopher W. Fraser and Eugene W. Myers. An editor for revision control. *ACM Transactions on Programming Languages and Systems*, 9(2):277–295, April 1987.

[8] International Business Machines. *Publication No. G221-2426: 3490 Magnetic Tape Subsystem Family*, 1996.

[9] L.A. Bjork, Jr. Generalized audit trail requirements and concepts for database applications. *IBM Systems Journal*, 14(3):229–245, 1975.

[10] Raymond A. Lorie. Physical integrity in a large segmented database. *IBM Transactions on Database Systems*, 2(1):91–104, March 1977.

[11] Peter B. Malcolm. *United States Patent No. 5,086,502: Method of Operating a Data Processing System*. Intelligence Quotient International, February 1992.

[12] Robert Morris. *United States Patent No. 5,574,906: System and method for reducing storage requirements in backup subsystems utilizing segmented compression and differencing*. International Business Machines, 1996.

[13] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[14] Dennis G. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(2):256–267, September 1976.

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

[16] V. P. Turnburke, Jr. Sequential data processing design. *IBM Systems Journal*, 2:37–48, March 1963.

[17] Joost M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, June 1978.

[18] VytalVault, Inc. *VytalVault Product Information.* http://www.vytalnet.com/vytalvlt/product.htm, 1996.

[19] Lin Yu and Daniel J. Rosenkrantz. A linear time scheme for version reconstruction. *ACM Transactions on Programming Languages and Systems*, 16(3):775–797, May 1994.