Versioned File Archiving, Compression, and Distribution

Josh MacDonald[1]

University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, Berkeley CA 94720, USA

Abstract. The XDELTA system implements a technique for archiving and compressing collections of many similar file versions. It stores only the differences between certain versions. I describe and discuss an algorithm for computing *file deltas*, present measurements, and demonstrate its application to versioned file-archival and efficient file-distribution network protocols.

1 Overview

The *file delta* problem is to compute a small set of instructions for transforming one file into another—one that is expected to be a function of the file's changes, not its content. This technique is well established for versioned file-archival. Though the advantages of using file deltas to transmit changes over a network are clear, specifying and widely deploying such a system efficient enough to justify itself is not as easy as it seems. There are a number of issues to overcome. First, the execution cost of computing and compressing deltas can be prohibitive–a site administrator might rather let everyone on the network suffer through a congested network condition than be CPU-bound while delivering efficient deltas to an uncongested network. Second, efficiently a integrating delta transmission into an application requires work; a server must store and name multiple versions of a file, compute deltas when requested, complicate existing protocols, and be fair—these are not always compatible.

Recently there have been advances in delta algorithms, but their use in network delta communication has been slow to follow. This work addresses many of these issues, and finishes with a few new challenges that I am currently working on. I have designed and implemented a decentralized scheme for highly-available, versioned file-archival, with which I have implemented a delta-based distribution and replication protocol built upon a simple, distributed file-archive abstraction. The system, called XDELTA, is implemented as the basis of multiple server, distributed version control in PRCS, the Project Revision Control System [10].

This paper outlines the XDELTA system in three sections: delta algorithms (§2), file-archival (§3), and distribution protocols (§4). Each section will detail related work, discuss previous implementations, outline a part of the system and measure it, and suggest future work.

2 Delta Algorithms

The *file delta problem* is to efficiently compute a delta d that can be stored compactly and used to construct a *To* file t^d from the set of k^d *From* files $F^d = \{f_0^d \dots f_{k^d}^d\}$. The *generate* operation computes d from F^d and t^d:

$$d = t^d \ominus F^d \tag{1}$$

while the *apply* operation computes the reverse:

$$t^d = d \oplus F^d \tag{2}$$

For our purposes, a file will be defined to be a sequence of bytes. For a file f, $size(f)$ denotes the length of f in bytes and set of files F, $size(F) = \sum_{f \in F} size(f)$. The notation, $f[i]$, denotes the byte at offset i of f, where $0 \leq i < size(f)$.

A good solution should be fast, perform well on large inputs, and expect to benefit from preprocessing F for repeated computations. Performance considerations for real files and memory systems make it desirable to operate in a single, sequential pass through t.

The *copy/insert* class of delta algorithms are those that produce a common delta encoding consisting of *copy* and *insert* operations. This class of algorithms is useful because there is a straightforward definition of an optimal solution, and the implementation of *apply* is extremely simple. For reasons to come, this type of delta will be represented in two separate parts: the *control* C^d and *insert-data* I^d. The control contains a sequence of instructions $c_1^d \ldots c_{z^d}^d$. The insert-data is represented as a file containing the concatenation of the data inserted by each insert instruction. Each instruction c_i is one of the two operations:

$$\texttt{copy}\ m\ o\ l \tag{3}$$

$$\texttt{insert}\ l \tag{4}$$

These are executed in order to construct the *To* file, where `copy` inserts the substring $f_m[o] \ldots f_m[o+l-1]$, and `insert` inserts the next l bytes from I^d.

The *insert/delete* class of algorithms is characterized by *insert* and *delete* instructions. Many existing tools for computing file deltas use this type of encoding, as it produces a form which is easy for humans to read.

Existing work on delta algorithms uses *paired* algorithms—those in which there is a single *From* file. For reasons which will become clear, the definition used here is more general than other presentations in allowing F to be a set of files rather than a single file. Of course, F can be reduced to a single file by concatenation, but attempting to generalize paired algorithms to multiple *From* files violates many of the assumptions made to improve these algorithms.

2.1 Related Work

Work on delta algorithms has been influenced by related work on the longest common subsequence problem, data compression, and pattern matching.

The first technique for computing file deltas used an insert/delete algorithm based upon finding the longest common subsequence (LCS) of two files. Among these, the UNIX `diff` utility was first used to list changes between two files in a line-by-line summary [11]. The `diff` utility is *line-oriented*—each insertion and deletion involves only complete lines. It was adopted by the early version control tools SCCS [15] and RCS [18]. Line-oriented algorithms, however, perform poorly on files which are not line terminated such as images, rich-text data, and object files.

Tichy developed a linear time and space, greedy copy/insert algorithm [17] using a suffix tree to search for matching substrings of the files in question. This simple greedy approach will be discussed next, and is easily shown to be optimal for copy/insert deltas under certain assumptions.

Burns [2] presents a recent summary of previous work and discusses several paired, copy/insert algorithms based upon a greedy approach using hashed Rabin-Karp fingerprints for string matching. He then refines it in several ways, because the simplest greedy algorithm requires space linear in the length of the file. He analyzes constant-space versions of this algorithm, making the assumption that changes between the files are the result of only insertions and deletions, for example. When F is a set of files, however, many of these improvements fail.

Hunt, Tichy, and Vo present an impressive paired delta algorithm called *vdelta* [9], which mixes Lempel-Ziv compression and differential compression to compute a compressed delta by using F as a dictionary to compress t. This particular approach is difficult to compare with others, because most delta algorithms compute uncompressed deltas.

A delta can be applied in-place if application can be performed using the space containing the data to which the delta is applied. An *insert/delete* delta can always be applied in-place, but an arbitrary *copy/insert* delta cannot. Burns and Long have shown techniques for computing *copy/insert* deltas that can be applied *in-place* [4].

2.2 Difference Metrics

Comparison of algorithms has been a difficult subject, as algorithms are studied variously in terms of delta size, compressed delta size, and optimality with a particular encoding. A solution that is optimal in one encoding cannot be easily compared with another encoding, without an absolute metric. Hunt, Tichy, and Vo suggest comparing delta algorithms with a difference metric based upon the length of the longest common subsequence [9]:

$$\mathit{difference} = \frac{\mathit{size}(f) + \mathit{size}(t)}{2} - \mathit{size}(\mathit{lcs}(f, t)) \tag{5}$$

where f is the single from input. The *difference* value is compared to the size of a delta produced by an algorithm, allowing its performance to be judged relative to results of other algorithms. This is troublesome for several reasons: there is no linear time and space algorithm for computing the exact LCS; it does not apply to multiple input files—since it does not allow permuted or repeated copy regions; and it is not even the most appropriate dynamic programming style of solution since it weighs an *insert* the same a *delete* instruction.

The copy/insert encoding is not optimal either; but it allows for permuted and repeated copied regions while the metric based on LCS does not. Additionally:

- An exact solution is known that requires $O(\mathit{size}(F))$ space [17]
- It characterizes non-paired delta algorithms
- The particular, uncompressed encoding is useful in practice

For these reasons, it seems appropriate to use the size of the unencoded, optimal *copy/insert* solution as a metric for comparing all delta algorithms. Also, using an encoding-independent metric allows algorithm designers to study encoding separately from search and pattern matching issues.

More precisely, the *copy/insert metric* m measures the upper bound on the size of the optimal sequence of *copy/insert* instructions, described next. Assuming an 8-bit byte, this metric estimates the total delta size in bits:

$$m(C^d, I^d) = \sum_{i=1}^{z^d} m(c_i^d) \tag{6}$$

$$m(\texttt{copy}\ m\ o\ l) = 1 + \lceil \lg l \rceil + \lceil \lg k^d \rceil + \lceil \lg o \rceil \tag{7}$$

$$m(\texttt{insert}\ l) = 1 + \lceil \lg l \rceil + 8l \tag{8}$$

Earlier work on copy/insert algorithms has used a simpler metric, treating all copies with unit cost. It then suffices to show that a simple greedy algorithm is optimal [2]. The unit-cost copy metric is not an accurate estimate of the file difference, so I propose the above. The greedy solution is not optimal for the refined metric.

2.3 The Greedy Algorithm for Copy/Insert Deltas

The greedy algorithm simply scans through the *To* file, finding the longest match at each position and generating a copy instruction. All unmatched strings produce an insert instruction.

ALGORITHM **A** *Compute* $(C^d, I^d) = t^d \ominus F^d$ *where* d *has a copy/insert encoding*

Assume that match $(m, o, l) = match(F, t, i)$, where the longest match to t starting at index i in F is $f_m[o] \ldots f_m[o+l-1]$. Another parameter, s, is used as a lower bound on the length of a copy instruction. For now, assume $s = 0$. Given these, a program to compute greedy copy/insert deltas is given in Figure 1.

```
generateDelta(t^d, F^d)
  i ← 0; z ← 0; I^d ← empty                          ▷ Step 1:  Initialize.
  while (i < size(t))                                  ▷ Step 2:  Loop.
    (m,o,l) ← match(F^d, t^t, i)
    if (l ≤ s)
      if (z > 0 and c_z^d = insert j)                  ▷ Step 3:  Insert.
        c_z^d ← insert j + 1
      else
        z ← z + 1; c_z^d ← insert 1
      I^d ← I^d append t^d[i]; i ← i + 1
    else
      z ← z + 1; c_z^d ← copy m o l; i ← i + 1       ▷ Step 4:  Copy.
  z^d ← z                                              ▷ Step 5:  Finish.
```

Fig. 1. Algorithm A

Steps 1 and 5 execute once in $O(1)$ steps. In the worst case, the loop beginning in step 2 is executed $size(t)$ times. Each execution of the loop requires $O(1)$ steps, plus the time taken by *match*. Altogether, he algorithm runs in $O(size(t) \cdot M)$ steps, where M is the time complexity of the *match* operation.

Burns [2] shows that this algorithm is optimal when copies are treated with unit cost. Burns and also Tichy [17] demonstrate various *match* functions that all require linear time and space. An algorithm requiring only constant space is required to operate on arbitrarily large files.

None of the available constant-space improvements, however, apply for multiple input files, which will be used for file archiving. As a result, XDELTA uses a linear-space approximation of the greedy algorithm.

2.4 The XDELTA Delta Algorithm

XDELTA uses a fast, linear time and space approximation to the greedy algorithm. The actual space required is chosen to be a fraction of the input size less than approximately 0.5. A *fingerprint* function is a hashing function for fixed-length strings for which collisions are unlikely. The algorithm uses a hashed index of the value of a fingerprint function at regular offsets in the *From* data to perform string matching. It then scans the *To* file performing the greedy algorithm. The approximation arises from hash and fingerprint collisions, and the approximate nature of the string-matching function.

For some constant s chosen to be a small power of 2, the algorithm computes a fingerprint on segments of length s in each file f at all offsets i divisible by s except, possibly, the last if s does not divide $size(f)$—in this case the segment is too short. The fingerprint function $a_{fi} = fingerprint(f, i)$ is evaluated in each file and inserted into a hash table, B, with hash function H. The number of such records is denoted $records(F) = \sum_{f \in F} \lfloor \frac{size(f)}{s} \rfloor$. The hash table does not chain hash collisions. Only a single (f, i) pair or *nil* is recorded for each bucket. The hash table is implemented as an array of 32-bit words of length $records(F)$. The *nil* value is encoded as the value 0, and (f, i) is mapped into the remaining values. Since i is bounded by $\lfloor \frac{size(f)}{s} \rfloor$, (f, i) can be easily fit into a 32-bit word provided that the number of files in F is less than s and the length of f is less than 2^{32}. Another array of words c_f is used for each file to index $c_f[i] = a_{fi}$ so that a fingerprint collision can be easily detected. The sum of lengths for these arrays is also $records(F)$, so in bytes the algorithm requires:

$$O\left(\sum_{f \in F} 8\lfloor\frac{size(f)}{s}\rfloor\right) \tag{9}$$

for the two arrays, where 8 is the sum of 4 bytes per 32-bit word for each of the separate data structures. The choice of s will be discussed later, but should be chosen to be at least 2^4, so the space required to run the algorithm is bounded by:

$$O\left(\frac{8 \cdot size(F)}{s}\right) \leq O\left(\frac{size(F)}{2}\right) = O(size(F)) \tag{10}$$

Algorithm B locates matching strings at each offset in the *To* file, supplying the match function used in algorithm A.

ALGORITHM **B** *Compute* $(m, o, l) = \text{match}(F, t, i)$.

```
match(t^d, F^d)
  x ← fingerprint(t, i)                                              ▷ Step 1:  Fingerprint.
  if (B[H(x)] = nil)                                                 ▷ Step 2:  Lookup.
    return no match
  else
    (m, o) ← B[H(x)]
  if (c_{f_m}[o] ≠ x) return no match                                ▷ Step 3:  Test for collision.
  l ← length of longest matching substring at offsets o and i        ▷ Step 4:  Grow.
  if (l < s) return no match
  return (m, o, l)
```

Fig. 2. Algorithm B

The fingerprint can be efficiently computed incrementally while advancing through the *To* file. The two characters entering and leaving the fingerprint range are used to advance the result in $O(1)$ steps. Locating string matches at each index takes, in the worst case, $O(s) = O(1)$ steps for a failed match and $O(size(t))$ for a successful one, but the total number of steps expended during all successful matches is bounded by $O(size(t))$. Using the result of §2.3, this gives a total time complexity of $O(size(t))$ for scanning the *To* input, plus $O(size(F))$ for constructing the index.

XDELTA uses the `adler32` fingerprint function originally used in gzip [19].

2.5 Analysis

A large value of s reduces the work done and space used by the algorithm, but will locate fewer matches. Too small a value of s can lead to copies so small that they they are less efficient than an equivalent insert encoding. This last consideration gives a lower bound on s:

$$m(\texttt{insert}\ s) > m(\texttt{copy}\ f\ o\ s) \tag{11}$$

$$8s > \lceil \lg k^d \rceil + \lceil \lg o \rceil \tag{12}$$

The right-hand-side of (12) is bounded by $\lceil \lg size(F) \rceil$, yielding $s > \frac{1}{8} \lceil \lg size(F) \rceil$.

Assuming the inputs, hash values, and fingerprint values have uniform distributions, the approximations made so far can be expected to perform well, as I will show. The key insight is that if a long match fails at one position due to a hashing collision, then it is likely to succeed at the next matching offsets. That is, a failure to match $t[i \ldots i+s-1]$ at $f_m[sk \ldots s(k+1)-1]$ because (m,k) is collides in the hash table is not a disaster because the next matching pair $t[i+s \ldots i+2s-1]$ and $f_m[s(k+1) \ldots s(k+2)-1]$ may succeed. The expected number of misses for a long match is proportional to the expected number of hash collisions, and therefore, the match is expected to fail asymptotically $O(1)$ times before the match is discovered. This causes an expected degradation that is linear only in the number of instructions.

The hash function and fingerprint function, composed, reduce segments in F to integers less than $records(F)$, which is chosen as the table size. When a conflict occurs during hashing, the algorithm must choose a winner. Several policies could be chosen. For example, a match appearing first in a file is potentially more valuable to the matching procedure since it may locate a match that contains colliding matches that occur later in the same file. Here, I will analyze a uniform random trial to decide whether to replace the incumbent record or not. By selecting the winner at random, the distribution of the hashing functions discussed above is not altered. Contrary to this analysis, however, simpler policies that never or always replace the colliding record seem to work equally well.

Another array $trials[i]$ is required to contain the number of collisions which have occured in the i^{th} bucket. The incumbent record or *nil* in bucket j is replaced with probability $P(replace) = \frac{1}{trials[j]+1}$. The additional array increases the space required by the algorithm in (10) by at most a factor of 3/2, since it brings the number of arrays from 2 to 3. The probability $P_k(i)$ represents the probability of the k^{th} record surviving the i^{th} trial. Then,

$$P_k(n) = \frac{1}{k} \cdot \frac{k}{k+1} \cdots \frac{n-2}{n-1} \cdot \frac{n-1}{n} = \frac{1}{n} \tag{13}$$

so the probability of survival is uniform.

The string matching function *fails* when a record that would have matched was discarded in a collision. The probability of failure is determined by assuming the hash functions behave randomly and computing the expected number of hash collisions. After hashing $M = records(F)$ records into N buckets, the expected number of of empty buckets is:

$$E(\#\ of\ empty\ buckets) = N\left(1 - \frac{1}{N}\right) \tag{14}$$

By choosing $N = M$, the expected number of empty buckets, and therefore collisions, is asymptotically $Ne^{-1} \approx 0.63N$. The expected number of trials before a successful match is found is the expected number of failures $E(failed) = \frac{e}{e-1}$. For files relatively large compared to s, the matching algorithm is expected to succeed after $\frac{e}{e-1}$ failures.

In addition to losing matches from hash collisions, matches are lost from only searching at a fraction of the offsets in F. For matches of length $l \geq s$, the match can be expected to be discovered at one of the $\lfloor \frac{l}{s} \rfloor$ interior points recorded in F. The first such point may be offset at most $s-1$ bytes from the true beginning, and the q^th such point is located at most $sq + s - 1$ bytes from the true beginning. From the above, we expect $q = \frac{e}{e-1}$, so the approximations produce matches which are expected to shorter than their true length by $\frac{se}{e-1} - s - 1$ bytes due to hash collisions. Thus, the expected penalty is linear in the number of copies, not the length of the inputs.

2.6 Inversion of Copy/Insert Deltas

Given file f and paired delta d that computes t from f, we would like to compute a delta d^{-1} that computes f from t. The solution will set up more general, direct delta manipulation in the next section. One might possess f and d, and want to inform a peer possessing t how to construct f by sending d^{-1}.

Algorithm **C** *Compute d^{-1} from d and f.*

I use a balanced interval tree to construct the reverse mapping for copy instructions in d. The tree supports insertion, deletion, and location of elements in $O(\lg V)$ steps for V, the number of intervals (see, for example, Cormen, Leiserson, and Rivest [5]). The program in Figure 3 computes the interval table R.

```
invertCopies (f, d)
  i ← 0                                  ▷ Current instruction in d.
  o_t ← 0                                ▷ Current offset in t.
  R ← empth                              ▷ Empty interval tree.
  while (i < z^d)
    if (c_i^d = copy f o l)
      R([o_t ... o_t + l)) ← [o ... o + l)
      o_t ← o_t + l;  i ← i + 1
    else (c_i^d = insert l)
      o_t ← o_t + l;  i ← i + 1
```

Fig. 3. Invert Copies in d from f

Using `invertCopies` to construct R, d^{-1} is constructed with an in-order traversal of R. Each range r records a mapping from $[o_f \ldots o_f + l)$ in f to $[o_t \ldots o_t + l)$ in t. Each such range may be used to construct `copy` $t\ o_t\ l$. Insert instructions are generated for all ranges not covered by R.

The performance of this algorithm depends on z^d, $z^{d^{-1}}$, and the $size(I^{d^{-1}})$. First, however, z^d and $z^{d^{-1}}$ can be related.

Theorem **1** *$z^{d^{-1}}$ is bounded according to:*

$$z^{d^{-1}} \leq 2z^d + 1 \tag{15}$$

Proof Let $z_i^{d^{-1}}$ by the number of instructions in d^{-1} as a function of the number of instructions i processed so far in d. Observe that $z_0^{d^{-1}} = 1$, corresponding to a single insert instruction in d^{-1}. In the worst case, each copied range may remove one existing instruction and create three; the induction hypothesis is $z_{i+1}^{d^{-1}} = 2 + z_i^{d^{-1}}$. □

The algorithm given does not produce an optimal delta, however. Its performance degrades when t's copied segments overlap in f–the previous analysis also suffers from repeated segments of length greater than s. Neglecting this case, the size of the delta is $size(d^{-1}) \approx size(f) - (size(t) - size(d))$.

The `invertCopies` procedure executes a loop z^d times, each requiring $O(\lg z^d)$ steps. The final construction, given R, requires $O(z^{d^{-1}} + size(I^{d^{-1}}))$ steps to compute the instructions in d^{-1}. Together, and using the results of theorem 1, the algorithm requires $O(z^d \lg z^d + size(I^{d^{-1}}))$. This time bound is acceptable, since z^d is expected to be small–it corresponds to the number of non-overlapping, longest matching substrings between t and f.

2.7 Applying Copy/Insert Deltas

A simple implementation of *apply* for copy/insert deltas is straightforward and requires $O(size(t))$ steps to apply d. The *reconstruct* operation takes a number of deltas and applies them all at once. Reconstruct accepts a tree of deltas in which each leaf is represented by a literal file and each internal node is represented by a delta, and constructs the file that the root node's delta constructs. It can be easily extended to operate on DAGs. Reconstruct can be considerably more efficient than simply applying each delta in sequence.

The technique for reconstruction resembles the inversion algorithm above. Suppose that t_r is being reconstructed from the delta tree rooted at d_r, where interior delta nodes are denoted d_i and leaf, literal nodes are denoted l_i. Algorithm D computes an interval tree mapping ranges in t_r to the literal data it is ultimately constructed from: files l_i and insert data I^{d_i}. It computes in a bottom-up traversal of the tree. A range $[l \ldots h)_{f,o_f}$ indicates the mapping from $[l \ldots h)$ to $[o_f \ldots o_f + h - l)$ in f, where f can also be the insert-data of some delta I_i^d, which is treated as just another literal file. Finally, the computed interval tree for d_r is used to write out the result in $O(size(t_r))$ steps.

Algorithm **D** *Reconstruct t_r from d_r.*

The program for computing t_r is given in Figure 4. The function `translate` translates the interval $[o \ldots o + l)$ from interval tree R_f to interval tree R_t. Each call to `reconstructRange` requires, in addition to the recursive calls, $O(z \lg z)$ steps and $O(z)$ space to insert z ranges into the resulting interval tree. For literal files, the call executes in constant time. For each delta d, z is bounded by z^d times the maximum of the number of ranges in each child's interval map.

The maximum number of ranges for delta d_1 due to the path $d_1 \rightarrow \ldots d_n \rightarrow l_i$ is $\prod_{i=1}^{n} z^i$, but is bounded by $O(size(t^{d_1}))$ since there are only so many intervals. The worst case occurs when a copy instruction copies from a range being constructed from the child delta which includes the range of every copy instruction in the child, and is unlikely in practice.

Let value z^* be the maximum such product for d_r. The intermediate storage required is the size of the largest interval table, which is $O(z^*)$. The time to reconstruct t_r from a tree of n deltas is $O(nz^* \lg z^* + size(t_r))$. This time complexity will be further examined for the archival scheme based upon it which follows.

Observe that this procedure allows a version to be partially extracted. The time complexity to extract a partial version of length p is still governed by $size(t)$ from the $z^* \lg z^*$ term, but only in the worst cast. Otherwise, the additional $size(t)$ term is reduced to $size(p)$, which is potentially significant.

2.8 Experimental Results

Several programs for computing file deltas are compared to evaluate their performance relative to the copy/insert metric m defined in §2.2. A total of 25584 file pairs that changed in the Linux kernel releases between versions 2.1.0 and 2.1.28 were used; both forward and reverse deltas were computed and the ratio of a program's delta size to the metric is plotted against the *To* file size in Figure 5. The results are averaged on the graph, with error bars indicating the standard deviation.

For the purposes of this abstract, only the results XDELTA and *vdelta* are shown.

For small files, *vdelta* outperforms XDELTA, but as the file size increases its performance degrades. In fact, XDELTA's relative performance continues to improve as input size grows, but the copy/insert metric was too expensive to compute for larger files so these results could not be included in the same plot. The performance of XDELTA and *vdelta* are, therefore, directly compared in Figure 6 for files from 25 to 30 megabytes in size. These files were the tarred contents of each of the Linux kernel releases used above. This test is especially relevant for determining how practical it is to release software package changes as a delta against the previous release, not on a file-by-file basis, but from the old package to the new package.

The linear-space approach has paid off on large input files. *Vdelta*'s delta size is close to the compressed size of the *To* file for many of the files in Figure 6, little space has been saved over simply compressing the *To* file. This is likely to be a result of the constant-space assumptions failing, and because the contents of the files may have been reordered.

Unfinished.

2.9 Future Work

Improvements in the space required by multi-input delta functions are necessary future work.

```
reconstruct(d_r)
  R = reconstructRange(d_r)
  for (increasing ranges r = [r_l ... r_h)_{l_r} in R)
    output the segment beginning at r_l in l_r
reconstructRange(d_i)                          ▷ for a delta
  R = empty interval tree
  o_i = 0                                      ▷ offset in I^{d_i}
  o_t = 0                                      ▷ offset in t^{d_i}
  for (f ∈ F^{d_i})
    intervals[f] = reconstructRange(f)
  for (each instruction c_i)
    if (c_i = copy f o l)
      translate (R, intervals[f], o, l)
      o_t = o_t + l
    else if (c_i = insert l)
      insert (R, [o_t ... o_t + l)_{I^{d_i}, o_i})
      o_i = o_i + l
      o_t = o_t + l
  for (f ∈ F^{d_i})
    free intervals[f]                          ▷ no longer required
reconstructRange(l_i)                          ▷ for a literal file
  R = empty interval tree
  insert (R, [0 ... size(l_i))_{l_i, 0})
  return R
```

Fig. 4. Reconstruct

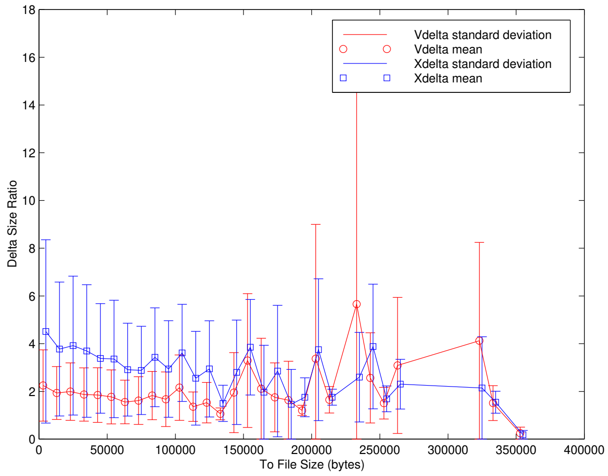

Fig. 5. Relative delta sizes.

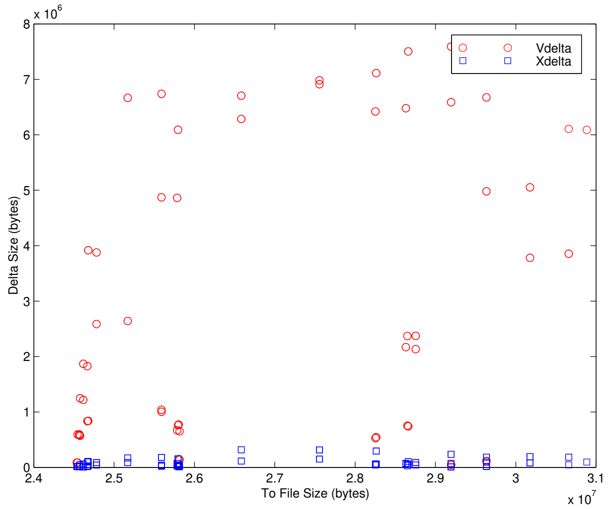

Fig. 6. Delta sizes for large files.

The reconstruct algorithm given has a similarity to the nearly linear time scheme presented for an editor with built-in version control by Fraser and Myers [6], though it does not directly apply.

The results achieved by the *vdelta* algorithm are very promising, but do not directly extend to all of the techniques presented here. Along these lines, the techniques presented below suggest that considering the delta problem in isolation is not the best way to proceed. There are strong connections to data compression and text retrieval, here; these can be taken advantage of when applying delta algorithms to particular application domains.

3 Versioned-File Archives

A versioned file archive supports the *get*, *put*, and *delete* operations. *Put* inserts a file into the archive, returning a name by which to retrieve it, and *get* retrieves the file. *Delete* indicates that a file will never be required again and that it's storage may be reclaimed.

The archive stores multiple versions of a *family* of files that are expected to be similar. Typically, an ancestry tree is imposed upon the archive; every version except the root has a parent. Changes between parent and child are stored as a delta. Requiring an application to know the ancestry proves difficult and inefficient, as this information is not always available and the storage model can lead to unpredictable performance. This mechanism has succeeded because the version control utilities which use it trusts users make the right decision by understanding the model, version naming conventions, and performance implications. Implementing efficient storage management on top of RCS, for example, proves difficult for these reasons [10].

The goal is to reduce the space required to store many versions in a family by taking advantage of their similarity and possibly their ancestry, while maintaining high performance and reliability for the *get*, *put*, and *delete* operations.

3.1 Related Work

Compact, versioned-file archiving can be achieved by storing literal files along with delta chains or trees, but other solutions are possible. Techniques for storing multiple versions have been studied and applied in several application areas: database management systems, programming language environments, version-control, and file-backup.

Both SCCS and RCS store an ancestry tree of versions where each vertex represents a version and each edge represents a delta. One version is stored as a literal file L and the rest are reconstructed by the application of deltas to the literal file. An archive using this organization often has unpredictable performance because the efficiency of an operation often depends on its distance from L.

SCCS doesn't actually store a literal file with separate deltas, but merges them together. Any version can be extracted in a single pass through the archive, collecting only those lines that are present in the version being extracted. This requires time linear in the size of the archive, including all deltas, not the file being extracted.

RCS improves upon this model by storing the literal file and deltas separately. The literal file is chosen to be a particular leaf node, causing retrieval and insertion on other branches to be much more expensive than on the distinguished *trunk*, which contains the literal file. Accessing any other branch requires the application of a series of deltas—the reconstruct problem. RCS builds a data structure representing a series of lines and applies each delta in succession. Let L be the sum of line counts for the deltas being applied. Reconstruct requires at least $O(L \lg L)$ steps using a balanced tree, though RCS in fact uses an $O(L^2)$ algorithm for simplicity[23].

Rosenkrantz and Yu demonstrate a reconstruct algorithm requiring $O(L)$ steps by computing a special form of the insert/delete delta [23].

Severance and Lohman show how to use file deltas for the storage of versions of large objects in databases and, more generally, for implementing transactions [16]. Alderson presents a space-efficient technique based on B-trees and file deltas that allows partial records to be retrieved without fully constructing the entire record, and for which the get operation has a predictable time-complexity for all versions [1]. Fraser and Myers demonstrate an editor with built-in version-control capabilities that uses AVL dags. It too has a nearly linear-time reconstruct operation [6]. These other storage schemes apply equally well to the file-archiving problem as presented, but constructing an archive out of deltas allows for the efficient transfer of archive components.

Burns and Long describe a technique for distributed backup based on delta compression [3]. Instead of a version tree, however, they use a technique called *version jumping* which involves storing a number of *clusters* consisting of one literal file and unit-length delta chains. This allows them to guarantee a file can be reconstructed with only two file accesses, which is important when slow storage is used. Analysis is performed to decide when a new literal file should be created, rather than a delta.

3.2 File Archiving With XDELTA

The XDELTA archive organization was originally designed as a replacement for RCS to be used in a distributed version control system. It stores a DAG of both literal files and deltas. Using an extension to the techniques discussed in §2, it also allows new versions to be efficiently transferred between archives as deltas. The internal organization is described here, and network communication is described in the next section.

XDELTA does not depend on a file's ancestry for organizational purposes. The archive consists only of a series of versions in increasing order of age. As branches of the family tree diverge, even though deltas between a version and its parent may be small, deltas between two leaves of the family tree may be quite large. Therefore, storing deltas in a simple series threatens to be quite inefficient since these *branch-crossing* deltas between leaves may require much more storage than the equivalent encoding using only deltas between parent and child. However, performance is critical, so storing versions as a simple series is appealing because it makes the time to retrieve a version to grow with its age, not as a function of its ancestry.

This suggests the organization of the XDELTA archive–simply a series *clusters* as in the *version-jumping* technique. The organization allows the time to retrieve any version to be bounded, while still addressing the diverging branch issue, using a combination of version-jumping and DAGs of deltas. When the time to retrieve the eldest version in a cluster reaches some threshold, its growth is halted. As with version-jumping, new literal files are occasionally introduced into storage to maintain several invariants.

When inserting a version, either a *forward* delta is inserted to produce the new version from the most recent literal file in the archive, or the new version becomes the literal file and a *reverse* delta replaces the

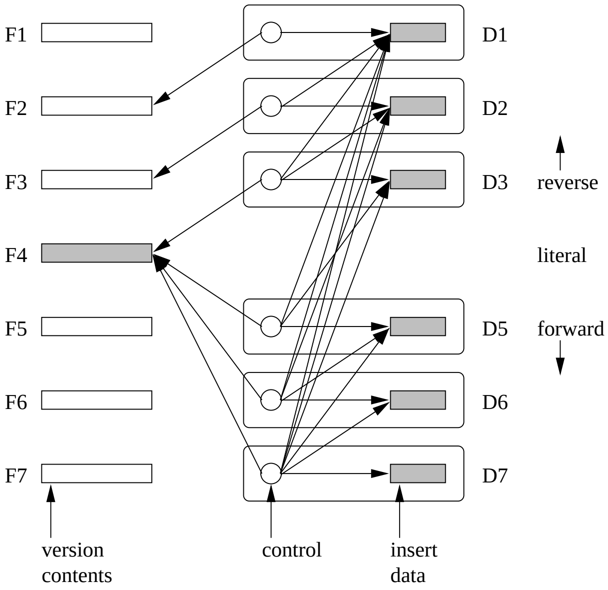

Fig. 7. Sample XDELTA Cluster

previous literal file. The decision to choose one over the other will be addressed later. In either case, the contents of the insert-data of recent deltas are used as additional sources for *copy* instructions for the new delta, as illustrated in Figure 7.

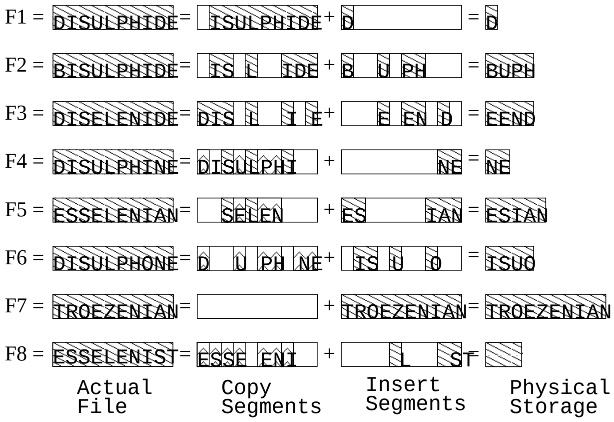

Fig. 8. Sample Deltas

Figure 8 illustrates the process on eight string versions $F1 \ldots F8$. The one literal file is $F7$. For demonstration, the example uses a simpler delta algorithm that only copies similar characters when they occur at the same offset in both files. In a realistic example, each character would also be represented by a longer, matching segment. The first column represents the full text of each string; these strings were derived according to the ancestry tree in Figure 9–each parent-child relation contains at least six matching characters.

When the first version is inserted into the archive, it becomes the literal file. Subsequent versions produce reverse deltas for some period of time until the time to extract the first file becomes excessive, at which

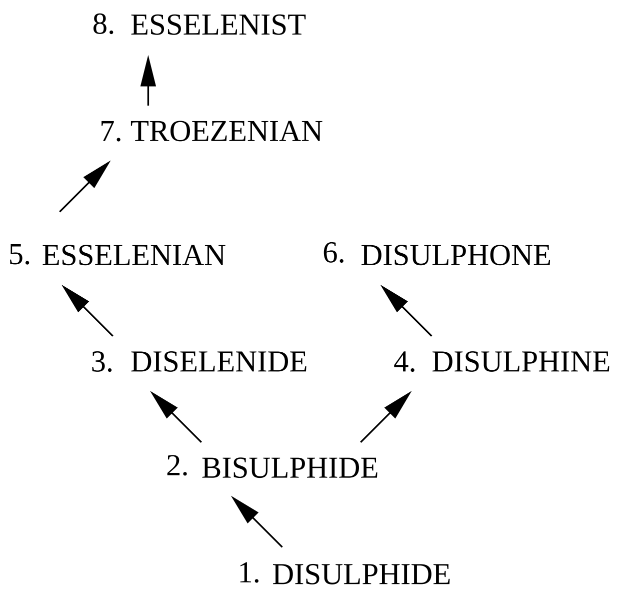

Fig. 9. String Relations in Figure 8

point the literal file is *frozen*. Forward deltas may then be inserted until the storage efficiency degrades, as measured by relative delta size, for example.

The time required to insert a version is always just the time to compute a single delta. Deletion requires $O(1)$ time. Retrieval is implemented with reconstruct, whose time complexity is $O(nz^* \lg z^* + size(t))$ to compute version t from n deltas, where z^* is bounded by the maximum product of the instruction counts on all paths from root to leaf and also by $size(t)$. Even assuming a small number of changes in each version, the time to retrieve a version is still $O(nsize(t))$, which is an unacceptable rate of growth, left unchecked. The maximum depth of a reverse delta DAG effects the time required to retrieve the oldest version in a cluster, and therefore determines when the literal file should be frozen.

Burns and Long show that version jumping does not result in a great loss of storage efficiency [3], and derive a worst-case analytic point at which storing a new literal file should be introduced as a function of the *compressibility*, or fractional size, of the delta. A simple heuristic for deciding to introduce a new literal file is used: introduce a new literal file when inserting a delta will increase the compression ratio of the cluster measured as the total size of all versions divided by the total size of all deltas and the literal file. The cycle of reverse deltas, followed by a literal file, followed by forward deltas, followed by a version jump repeats indefinitely.

In Figure 8, the columns illustrate the relation between each file and the delta that constitutes it. In the control and insert-data columns, a shaded region indicates that the text is either copied or inserted at that position in the file. In the control column, two shadings are used to distinguish copies from the subsequent literal file and copies from previous deltas. For instance, the characters `EN` in `ESSELENIAN` are copied from the insert-data segment of the delta representing `DISELENIDE`. The `DIS` in `DISELENIDE` is copied from the literal file `DISULPHONE` which follows it. This illustrates how using previous deltas handles diverging files. The intervening insertion of `DISULPHINE`, on a different branch, does not disturb the encoding efficiency. The number of previous deltas used must be somehow chosen and limited in number. A window size w is chosen according to available space. Increasing w promises more available matches, but at the same time unrelated data threatens to evict matching data from the hash table. The previous deltas chosen are the most recent, but other strategies for locating possibly related files can be used. The ancestry of a version can be used, in addition, to select the most recent delta computed *to* an ancestor of the one being inserted, it is likely related.

In addition to these properties,

- XDELTA uses an efficient, non-line-oriented delta mechanism
- The archive can trade time for space by precomputing and adjusting the delta DAG depth
- The number of deltas that must be applied to retrieve a version is proportional to the number of put operations since that version was created; RCS does not have this property, for example
- Knowledge of a versions ancestry is not required for organizational purposes
- Partial compression is possible–older versions may be compressed without affecting the retrieval time of newer versions, and when combined with precomputed indices, they may be lazily uncompressed
- Obsolete deltas may be simply deleted as soon as no other live deltas reference their insert-data.

3.3 Experimental Results

Unfinished.

3.4 Future Work

Delta compression techniques have proven successful, but archival techniques have received little exploration. As computers become more and more dependent upon network communication and distributed data, distributed file-archival techniques will increase in importance.

A popular method for viewing the history of a version-controlled document involves attributing each line with the name of the author and how recently it was changed or introduced. Directly constructing such an annotation from copy/insert deltas is not as straight-forward, and more difficult to present since it is not line-oriented. Another type of annotation allows a file to be viewed in a particular context, allowing not only how each line was authored, but the author of the next modification or deletion. More advanced tools for viewing (or querying) relations between files in a file-archive are needed.

The windowing technique technique presented for determining a set of input files could be replaced with a coarse text retrieval algorithm to determine the set of files likely to contain substrings of the output file.

4 Distribution Protocols

There is a broad class of applications that operate on large quantities of data, often highly similar or version controlled, and need to move that data across slow networks frequently. Delta compression can be used both for versioned file archival, for reducing network transmission, as a network data cache. The effort required to implement these operations efficiently, however, is considerable, and has hindered advances so far. In this section, I propose a simple, yet powerful, abstraction for the purposes of constructing application protocols, according to the following considerations:

- Data should be named by content. The MD5 checksum [14], for example, can be used to name a file; doing so removes the need for any global consistency.
- Versions are read-only and should be highly available; therefore replication is required
- Disk space is more available than network bandwidth
- A moderate amount of computation for each client connection is permitted, but should be minimized
- Compressing data on a per-connection, per-transfer basis heavily loads a server, and is undesirable, as evidenced by the maintainers of popular CVS servers [22]
- Each server should be autonomous, so there are no global consistency issues; deltas must be receivable out of order
- Servers must be allowed to store disjoint sets of files
- Servers should transfer files using a minimum of network bandwidth
- Pairwise peer synchronization should be used to exchange archive contents; at the application level, synchronization can be pushed or pulled, since the protocol is symmetric

– A server should be easily usable as a proxy-cache.

A scheme that allows the direct exchange of pre-compressed deltas is desirable. Each archive maintains a set of versions for each family in a hierarchically named set of file families. The application is responsible for deciding which files should be stored at each server. The underlying storage mechanism only provides efficient delta-storage and push/pull delta-transfer. Aside from efficiency issues, this type of service resembles the operation of NNTP servers for news transport as described in RFC 1036 [8]. This is a nice abstraction for building application servers; an application may co-reside with a distribution server, leaving the details of efficient transfer to the abstraction and only implementing application specific control and policy. Since the service can be easily proxied, it reside on a client machine or a slow network boundary. This allows all the details of efficient transport to be left to a decentralized, and thus locally administered, service, so client to server protocols need not be effected.

All of this requires a small modification to the copy/insert delta format, described next.

4.1 Normalized Copy/Insert Deltas

According to the above list of considerations, each server may store a different set of files, and have received them in a different order from everyone else. Deltas depending on the insert-data of other deltas, however, cannot be easily exchanged because a server's peer is not guaranteed to have the same set of deltas, or to have received its versions in the same order. A *normalized* copy/insert delta includes additional information to avoid this problem. The header for delta d names each dependent file $f_i^d \in F^d$ by its MD5 checksum. Additionally, when f_i^d is the insert-data of delta d_i, it also includes the MD5 checksum of the file constructed by d_i, t^{d_i}. An alternate sequence of copy instructions is included for each copy instruction in d_i, allowing the copy to be performed either from the insert-data of the delta or the literal file, whichever is available. This allows, in many cases, a server to send its peer a version by simply sending its existing delta, perhaps pre-compressed.

These modifications increases the cost of a copy instruction by a small constant factor, except in pathological cases where one version contains literal copies of another delta in the archive. A single copy instruction may be followed by one or more normalized copy instructions because copies from the insert-data of a delta may be fragmented in the file it constructs.

4.2 Related Work

Douglis and others have studied the addition of delta compression to `http` [12]. Their results are encouraging, showing substantial decreases in both end-to-end latency and network transmission for several delta transfer strategies. There is an official W3 Consortium proposal for the distribution and replication of `http` [20]. It includes the specification of a standard copy/insert delta format for exchange [21]. The protocol includes extension for replicating content versions using delta transfers. Content is also named by a configurable message digest, such as MD5. This work has not dealt directly with implementation issues, but has specified an exchange protocol.

The *rsync* algorithm, due to Tridgell and Mackerras, is for remotely synchronizing the contents of a pair of files [19]. It does this using a computation similar to algorithm A, but transfers a table of checksums for one file to the peer who performs string matching in the other file using MD5 checksums rather than direct string comparison. The peer then replies with a request for the list of blocks that it is missing. This is quite effective for synchronizing files between two peers that do not share any versions in common.

CVSup [13] is a distribution protocol for CVS that globally synchronizes the contents of client CVS repositories with a mirror or the master repository. This is accomplished primarily by decomposing RCS files in the repository and transferring only the deltas that are missing on the client side. This model requires centralized control.

4.3 A Distributed Archive

A distributed archive consists of a network of autonomous archives as described above. An application is responsible for controlling each archive appropriately. I'll analyze a simple arrangement first, as it demonstrates an important special case. Once the simple arrangement has been described, I will extend the results for less constrained circumstances.

Suppose that for a file family being archived, the policy of the application running on client S_c is to replicate every version in server S_s. Then S_s can efficiently send S_c new versions as a delta with little computation. Only the literal file, for which no delta is immediately available, requires processing, and the result can be cached. If the version being sent is already available as a delta, it can just be sent. If the distributed storage can be arranged as a tree in which each child replicates at least its parent's set of versions, this situation applies.

All of this relies on the normalized delta–it allows copies from both a *site-specific*, readily available segment and a *universal*, but perhaps not physically present, segment. When transferring a delta the site-specific segment may be missing at the receiver. All the sender need do is verify that the receiver can construct each of the necessary universal segments, allowing it to internalize the delta by translating it to depend upon its own site-specific segments. The client, therefore, is responsible for the majority of the work required to receive. Suppose that before synchronization S_c contains the set of versions F_c, and S_s contains the set of versions F_s. S_c requests a transfer of the difference $F_t = F_s - F_c$.

Claim 1 *For each version f to be sent, if S_s is storing f as delta d_f it can simply send d_f. Otherwise f is stored as a literal file, and S_s can compute and send a forward delta $f \ominus f_c$, for some recent $f_c \in F_c$.*

Proof Each delta that was sent must be applicable by the receiver after the complete transmission. Since the client requested all versions, those that were stored and sent as deltas depend on either a version that the client already has, one of the other deltas that was sent, or a literal file for which a forward delta was sent. Since the forward deltas that were computed depend only on versions already in F_c, no cyclic dependencies are introduced at the client. □

Assuming archives synchronize with each other with period p and versions are deleted when their age surpasses a, then an archive may pre-compute only a single, forward delta for each literal file in the archive. The *From* input for this computation should be the first version whose age is greater than p and less than $a - p$. Each server can also keep a record of its peers' contents to allow incremental log updates and each archive to compute a forward delta that the greatest number of its peers will be capable of applying.

4.4 Arbitrary Policies

The previous section dealt only with the case where the client's policy is to archive every file archived by the server. This special case allows many new deltas to be sent uninterpreted because the client replicates every version in the server. The case of a literal file is no different, but when a server must send an existing delta to a peer replicating fewer versions than itself, it must translate out the peer's missing references before sending. Suppose that S_c requests F_t, where $F_t \cup F_c$ is a subset of the server's set of files F_s and therefore S_s may not be able to directly send a delta.

Computing such a delta is essentially similar to the reconstruct operation. First, `reconstructRange` is used to construct an interval tree R. Each copied range in R is used to either construct a copy instruction, if S_s has the necessary reference, or is translated into an insert instruction if S_s does not have the reference. This procedure reduces to the above special case when the client has requested every file that the server contains. Again, the time complexity does not directly depend on the size of the version involved, but on the number of changes and deltas between them.

4.5 Receiving Normalized Deltas

It is guaranteed that when an archive receives a delta from a peer, it can apply that delta, though not as efficiently as possible. In fact, `reconstruct` will operate as given by simply using the normalized copy indices when a segment is missing. However, the reconstruct algorithm given assumed that all cross-edges in the DAG's depth-first-subtree were directed at literal files, allowing the recursive algorithm to free a node's interval table immediately after its parent has used it. Foreign deltas do not maintain this invariant; so an efficient implementation of `reconstructRange` must instead compute interval trees in a bottom-up DAG traversal; each node's interval tree can be discarded after each parent's interval-tree has been computed. This increases the space required by the algorithm by, at most, a factor of 2, since one additional interval-tree may required.

Given an implementation of `reconstructRange`, a delta can be internalized by computing the range table mapping copy ranges to local segment ranges, and re-generated as in algorithm C.

4.6 Future Work

I have implemented the system described as the basis of a distributed version-control system. The synchronize operation is extended to use the *rsync* algorithm when peers share no files in common, and to send the literal file when one peer contains no files. Future work will include a complete experimental analysis of the system's performance and behavior. Making such an archive operate securely and with minimum trust requirement is a future challenge.

5 Conclusion

The file delta problem and several applications have been examined. Many applications desires both compression, differential compression, and other features which complicate the direct application of available delta algorithms. These additional considerations influenced the design of a new system with a new delta algorithm.

The XDELTA delta algorithm requires linear space, but requires less space than the size of the inputs, meaning the algorithm is feasible for files which are required to reside in memory by other applications. The algorithm trades space to improve delta size, and approximates the exact algorithm to reduce execution space and time. It handles multiple inputs, a subject which has not previously been studied, and performs especially well on large inputs.

The XDELTA archiving scheme was designed as a replacement for RCS with network distribution in mind. It replaces the ancestry tree with a serial model using a combination of version-jumping, forward deltas, and reverse deltas. Multiple *From* inputs are used when computing deltas, allowing the contents of more than one previous version to be referenced by copy instructions.

Finally, a scheme for decentralized network file archiving was presented. The copy/insert delta encoding is altered so that each copy instruction is replaced by two sets of instructions, allowing a two machines to directly exchange deltas without any computation in some cases. I propose the distributed file archive as a abstraction for efficiently implementing certain types of distributed applications, such as version-control, software-update, groupware, and `http` caching and replication.

References

1. ALDERSON, A. A space-efficient technique for recording versions of data. *Software Engineering Journal 3*, 6 (Nov. 1988), 240–246.
2. 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.
3. 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).
4. BURNS, R. C., AND LONG, D. D. E. In-place reconstruction of delta compressed files. In *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing* (June 1998).
5. CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms.* MIT Press, 1993.

6. 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.
7. Horton, M. R. RFC 850: Standard for interchange of USENET messages, June 1983. Obsoleted by RFC1036 [8]. Status: UNKNOWN.
8. Horton, M. R., and Adams, R. RFC 1036: Standard for interchange of USENET messages, Dec. 1987. Obsoletes RFC0850 [7]. Status: UNKNOWN.
9. Hunt, J. J., Vo, K.-P., and Tichy, W. F. An empirical study of delta algorithms. *Lecture Notes in Computer Science 1167* (July 1996), 49–66.
10. MacDonald, J., Hilfinger, P. N., and Semenzato, L. Prcs: The project revision control system. *Lecture Notes in Computer Science 1439* (July 1998), 33–45.
11. Miller, W., and Myers, E. W. A file comparison program. *Software—Practice and Experience 15*, 11 (Nov. 1985), 1025–1040.
12. 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.
13. Polstra, J. Program source for CVSup. ftp://ftp.cvsup.freebsd.org/pub/CVSup, 1996.
14. Rivest, R. RFC 1321: The MD5 message-digest algorithm, Apr. 1992. Status: INFORMATIONAL.
15. Rochkind, M. J. The source code control system. *IEEE Transactions on Software Engineering 1*, 4 (Dec. 1975), 364–370.
16. Severance, D. S., and Lohman, G. M. Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems* (Sept. 1976), 256–267.
17. Tichy, W. F. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), 309–321.
18. Tichy, W. F. RCS: A system for version control. *Software—Practice and Experience 15*, 7 (July 1985), 637–654.
19. Tridgell, A., and Mackerras, P. The Rsync algorithm. Tech. Rep. TR-CS-96-05, The Australian National University, June 1996.
20. van Hoff, A., Giannandrea, J., Hapner, M., Carter, S., and Medin, M. The http distribution and replication protocol. http://www.w3.org, Aug. 1997.
21. van Hoff, A., and Payne, J. Generic diff format specification. http://www.w3.org, Aug. 1997.
22. Webmaster. Mozilla cvs usage instructions. http://www.mozilla.org/cvs.html.
23. 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.