Experimentally Evaluating In-Place Delta Reconstruction

Randal Burns
Dept. of Computer Science
Johns Hopkins Univ.
randal@cs.jhu.edu

Larry Stockmeyer
Dept. of Computer Science
IBM Almaden Research Center
stock@almaden.ibm.com

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

Abstract

In-place reconstruction of delta compressed data allows information on devices with limited storage capability to be updated efficiently over low-bandwidth channels. Delta compression encodes a version of data compactly as a small set of changes from a previous version. Transmitting updates to data as delta versions saves both time and bandwidth. In-place reconstruction rebuilds the new version of the data in the storage or memory space the current version occupies – no additional scratch space is needed. By combining these technologies, we support large-scale, highly-mobile applications on inexpensive hardware.

We present an experimental study of in-place reconstruction algorithms. We take a data-driven approach to determine important performance features, classifying files distributed on the Internet based on their in-place properties, and exploring the scaling relationship between files and data structures used by in-place algorithms. We conclude that in-place algorithms are I/O bound and that the performance of algorithms is most sensitive to the size of inputs and outputs, rather than asymptotic bounds.

1 Introduction

We develop algorithms for data distribution and version management to be used for highly-mobile and resource-limited computers over low-bandwidth networks. The software infrastructure for Internet-scale file sharing is not suitable for this class of applications, because it makes demands for network bandwidth and storage/memory space that many small computers and devices cannot meet.

While file sharing is proving to be the new prominent application for the Internet, it is limited in that data are not writable nor are versions managed. The many recent commercial and freely available systems underscore this point, examples include Freenet [1] and GnuTella [2]. Writable replicas greatly increase the complexity of file sharing – problems include update propagation and version control.

Delta compression has proved a valuable tool for managing versions and propagating updates in distributed systems and should provide the same benefits for Internet file sharing. Delta-compression has been used to reduce latency and network bandwidth for Web serving [4, 20] and backup and restore [6].

Our in-place reconstruction technology addresses one of delta compression's major shortcomings. Delta compression makes memory and storage demands that are not reasonable for low-cost, low-resource devices and small computers. In-place reconstruction allows a version to be updated by a delta in the memory or storage that it currently occupies; reconstruction needs no additional scratch space or space for a second copy. An in-place reconstructible delta file is a permutation and modification of the original delta file. This conversion comes with a small compression penalty. In-place reconstruction brings the latency and bandwidth benefits of delta compression to the space-constrained, mass-produced devices that need them the most, such as personal digital assistants, cellular phones, and wireless handhelds.

A distributed inventory management system based on mobile-handheld devices is an archetypal application for in-place technology. Many limited-capacity devices track quantities throughout an enterprise. To reduce latency, these devices cache portions of the database for read-only and update queries. Each device maintains a radio link to update its cache and run a consistency protocol. In-place reconstruction allows the devices to keep their copies of data consistent using delta compression without requiring scratch space, thereby increasing the cache utilization at target devices. Any available scratch space can be used to reduce compression loss, but no scratch space is required for correct operation. We observe that in-place reconstruction applies to both structured data (databases) and unstructured data (files), because they manipulate a delta encoding, as opposed to the original data. While algorithms for delta compressing structured data are different [9], they employ encodings that are suitable for in-place techniques.

1.1 Delta Compression and In-Place Reconstruction

Recent developments in portable computing and computing appliances have resulted in a proliferation of small network attached computing devices. These include personal digital assistants (PDAs), Internet set-top boxes, network computers, control devices, and cellular devices. The data contents of these devices are often updated by transmitting the new version over a network. However, low bandwidth channels and heavy Internet traffic often makes the time to perform software update prohibitive.

Differential or delta compression [3, 13, 9, 8], encoding a new version of a file compactly as a set of changes from a previous version, reduces the size of the transmitted file and, consequently, the time to perform software update. Currently, decompressing delta encoded files requires scratch space, additional disk or memory storage, used to hold a second copy of the file. Two copies of the file must be available concurrently, as the delta file reads data from the old file version while materializing the new file version in another region of storage. This presents a problem because network attached devices often cannot store two file versions at the same time. Furthermore, adding storage to network attached devices is not viable, because keeping these devices simple limits their production costs.

We modify delta encoded files so that they are suitable for reconstructing the new version of the file *in-place*, materializing the new version in the same memory or storage space that the previous version occupies. A delta file encodes a sequence of instructions, or *commands*, for a computer to materialize a new file version in the presence of a *reference* version, the old version of the file. When rebuilding a version encoded by a delta file, data are both copied from the reference version to the new version and added explicitly when portions of the new version do not appear in the reference version.

If we were to attempt naively to reconstruct an arbitrary delta file in-place, the resulting output would often be corrupt. This occurs when the delta encoding instructs the computer to copy data from a file region where new file data has already been written. The data the algorithms reads have already been altered and the algorithm rebuilds an incorrect file.

We present a graph-theoretic algorithm for modifying delta files that detects situations where a delta file attempts to read from an already written region and permutes the order that the algorithm applies commands in a delta file to reduce the occurrence of conflicts. The algorithm eliminates the remaining conflicts by removing commands that copy data and adding explicitly these data to the delta file. Eliminating data copied between versions increases the size of the delta encoding but allows the algorithm to output an in-place reconstructible delta file.

Experimental results verify the viability and efficiency of modifying delta files for in-place reconstruction. Our findings indicate that our algorithm exchanges a small amount of compression for in-place reconstructibility.

Experiments also reveal an interesting property of these algorithms that conflicts with algorithmic analysis. We show in-place reconstruction algorithms to be I/O bound. In practice, the most important performance factor is the output size of the delta file. This means that heuristics for eliminating data conflicts that minimize lost compression are superior to more time efficient heuristics that lose more compression. Any time saved in detecting and eliminating conflicts is lost when writing a larger delta file out to storage.

2 Related Work

Encoding versions of data compactly by detecting altered regions of data is a well known problem. The first applications of delta compression found changed lines in text data for analyzing the recent modifications to files [11]. Considering data as lines of text fails to encode minimum sized delta files, as it does not examine data at a fine *granularity* and finds only matching data that are *aligned* at the beginning of a new line.

The problem of representing the changes between versions of data was formalized as string-to-string correction with block move [24] – detecting maximally matching regions of a file at an arbitrarily fine granularity without alignment. However, delta compression continued to rely on the alignment of data, as in database records [23], and the grouping of data into block or line granularity, as in source code control systems [22, 25], to simplify the combinatorial task of finding the common and different strings between versions.

Efforts to generalize delta compression to un-aligned data and to minimize the granularity of the smallest change resulted in algorithms for compressing data at the granularity of a byte. Early algorithms were based upon either dynamic programming [19] or the greedy method [24, 21, 17] and performed this task using time quadratic in the length of the input files.

Delta compression algorithms were improved to run in linear time and linear space. Algorithms with these properties have been derived from suffix trees [27, 18, 16] and as a generalization of Lempel-Ziv data compression [12, 13, 8]. Like algorithms based on greedy methods and dynamic programming, these algorithms generate optimally compact delta encodings.

Recent advances produced algorithms that run in linear time and constant space [3]. These differencing algorithms trade a small amount of compression, verified experimentally, in order to improve performance.

Any of the linear run-time algorithms allow delta compression to scale to large input files

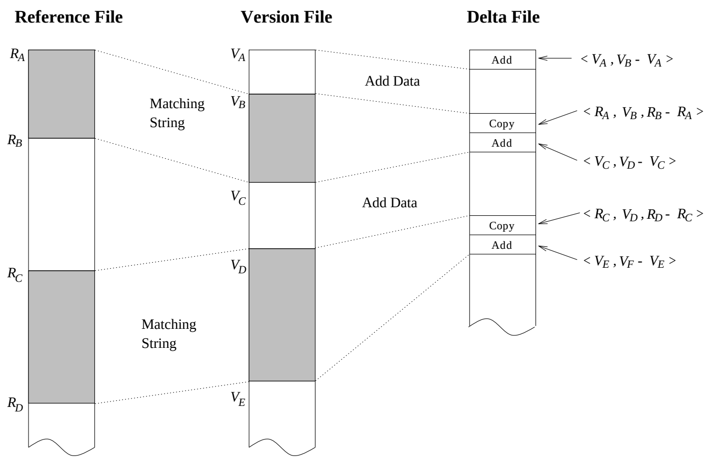


Figure 1: Encoding delta files. Common strings are encoded as *copy* commands $\langle f, t, l\rangle$ and new strings in the new file are encoded as *add* commands $\langle t, l\rangle$ followed by the string of length l of added data.

without known structure and permits the application of delta compression to file system backup and restore [6].

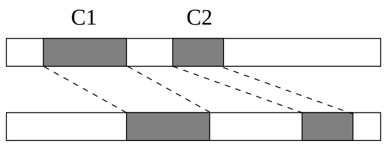
Recently, applications distributing HTTP objects using delta files have emerged [20, 4]. This permits web servers to both reduce the amount of data transmitted to a client and reduce the latency associated with loading web pages. Efforts to standardize delta files as part of the HTTP protocol and the trend toward making small network devices HTTP compliant indicate the need to distribute data to network devices efficiently.

3 Encoding Delta Files

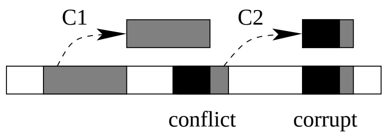
Differencing algorithms encode the changes between two file versions compactly by finding strings common to both versions. We term these files a *version file* that contains the data to be encoded and a *reference file* to which the version file is compared. Differencing algorithms encode a file by partitioning the data in the version file into strings that are encoded using copies from the reference file and strings that are added explicitly to the version file (Figure 1). Having partitioned the version file, the algorithm outputs a delta file that encodes this version. This delta file consists of an ordered sequence of *copy* commands and *add* commands.

An *add* command is an ordered pair, $\langle t, l\rangle$, where t (to) encodes the string offset in the file version and l (length) encodes the length of the string. The l bytes of data to be added follow the command. A *copy* command is an ordered triple, $\langle f, t, l\rangle$ where f (from) encodes the offset in the reference file from which data are copied, t encodes the offset in the new file where the data are to be written, and l encodes the length of the data to be copied. The *copy* command moves the string data in the interval $[f, f + l - 1]$ in the reference file to the interval $[t, t + l - 1]$ in the version file.

In the presence of the reference file, a delta file rebuilds the version file with *add* and *copy* commands. The intervals in the version file encoded by these commands are disjoint. Therefore, any permutation of the command execution order materializes the same output version file.



(a) Delta copy



(b) In-place copy

Figure 2: Data conflict and corruption when performing copy command C1 before C2.

4 In-Place Modification Algorithms

An in-place modification algorithm changes an existing delta file into a delta file that reconstructs correctly a new file version in the space the current version occupies. At a high level, our technique examines the input delta file to find *copy* commands that read from the write interval (file address range to which the command writes data) of other *copy* commands. The algorithm represents potential data conflicts in a digraph. The algorithm topologically sorts the digraph to produce an ordering on *copy* commands that reduces data conflicts. We eliminate the remaining conflicts by converting *copy* commands to *add* commands. The algorithm outputs the permuted and converted commands as an in-place reconstructible delta file. Actually, as described in more detail below, the algorithm performs permutation and conversion of commands concurrently.

4.1 Conflict Detection

Since we reconstruct files in-place, we concern ourselves with ordering commands that attempt to read a region to which another command writes. For this, we adopt the term *write before read* (*WR*) conflict [5]. For *copy* commands $\langle f_i, t_i, l_i \rangle$ and $\langle f_j, t_j, l_j \rangle$, with $i < j$, a *WR* conflict occurs when

$$[t_i, t_i + l_i - 1] \cap [f_j, f_j + l_j - 1] \neq \emptyset. \tag{1}$$

In other words, *copy* command i and j conflict if i writes to the interval from which j reads data. By denoting, for each *copy* command $\langle f_k, t_k, l_k \rangle$, the command's read interval as $\textit{Read}_k = [f_k, f_k + l_k - 1]$ and its write interval as $\textit{Write}_k = [t_k, t_k + l_k - 1]$, we write the condition (1) for a WR conflict as $\textit{Write}_i \cap \textit{Read}_j \neq \emptyset$. In Figure 2, commands C1 and C2 executed in that order generate a data conflict (blacked area) that corrupts data when a file is reconstructed in place.

This definition considers only *WR* conflicts between *copy* commands and neglects *add* commands. *Add* commands write data to the version file; they do not read data from the reference file. Consequently, an algorithm avoids all potential *WR* conflicts associated with adding data by placing *add* commands at the end of a delta file. In this way, the algorithms completes all reads associated with *copy* commands before executing the first *add* command.

Additionally, we define *WR* conflicts so that a *copy* command cannot conflict with itself. Yet, a single *copy* command's read and write intervals intersect sometimes and would seem to cause a conflict. We deal with read and write intervals that overlap by performing the copy in a *left-to-right* or *right-to-left* manner. For command $\langle f, t, l \rangle$, if $f \geq t$, we copy the string byte by byte starting at the left-hand side when reconstructing the original file. Since, the f (from) offset always exceeds the t (to) offset in the new file, a *left-to-right* copy never reads a byte over-written by a previous byte in the string. When $f < t$, a symmetric argument shows that we should start our copy at the right hand edge of the string and work backwards. For this example, we performed the copies in a byte-wise fashion. However, the notion of a left-to-right or right-to-left copy applies to moving a read/write buffer of any size.

To avoid *WR* conflicts and achieve the in-place reconstruction of delta files, we employ the following three techniques.

1. Place all *add* commands at the end of the delta file to avoid data conflicts with *copy* commands.
2. Permute the order of application of the *copy* commands to reduce the number of write before read conflicts.
3. For remaining *WR* conflicts, remove the conflicting operation by converting a *copy* command to an *add* command and place it at the end of the delta file.

For many delta files, no possible permutation eliminates all *WR* conflicts. Consequently, we require the conversion of *copy* commands to *add* commands to create correct in-place reconstructible files for all inputs.

Having processed a delta file for in-place reconstruction, the modified delta file obeys the property

$$(\forall j)\left[\mathit{Read}_j \cap \left(\bigcup_{i=1}^{j-1} \mathit{Write}_i\right) = \emptyset\right], \tag{2}$$

indicating the absence of *WR* conflicts. Equivalently, it guarantees that a *copy* command reads and transfers data from the original file.

4.2 CRWI Digraphs

To find a permutation that reduces *WR* conflicts, we represent potential conflicts between the *copy* commands in a digraph and topologically sort this digraph. A topological sort on digraph $G = (V, E)$ produces a linear order on all vertices so that if G contains edge $\vec{uv}$ then vertex u precedes vertex v in topological order.

Our technique constructs a digraph so that each *copy* command in the delta file has a corresponding vertex in the digraph. On this set of vertices, we construct an edge relation with a directed edge $\vec{uv}$ from vertex u to vertex v when *copy* command u's read interval intersects *copy* command v's write interval. Edge $\vec{uv}$ indicates that by performing command u before command v, the delta file avoids a WR conflict. We call a digraph obtained from a delta file in this way a *conflicting read write interval* (*CRWI*) digraph. A topologically sorted version of this graph adheres to the requirement for in-place reconstruction (Equation 2).

4.3 Strategies for Breaking Cycles

As total topological orderings are possible only on acyclic digraphs and CRWI digraphs may contain cycles, we enhance a standard topological sort to break cycles and output a total topological order on a *subgraph*. Depth-first search implementations of topological sort [10] are modified easily to detect cycles. Upon detecting a cycle, our modified sort breaks the cycle by removing a vertex. When completing this enhanced sort, the sort outputs a digraph containing a subset of all vertices in topological order and a set of vertices that were removed. This algorithm re-encodes the data contained in the *copy* commands of the removed vertices as *add* commands in the output.

As the string that contains the encoded data follows converted *add*, this replacement reduces compression in the delta file. We define the amount of compression lost upon deleting a vertex to be the *cost* of deletion. Based on this cost function, we formulate the optimization problem of finding the minimum cost set of vertices to delete to make a digraph acyclic. A *copy* command is an ordered triple $\langle f, t, l \rangle$. An *add* command is an ordered double $\langle t, l \rangle$ followed by the l bytes of data to be added to the new version of the file. Replacing a *copy* command with an *add* command increases the delta file size by $l - \|f\|$, where $\|f\|$ denotes the size of the encoding of offset f. Thus, the vertex that corresponds to the copy command $\langle f, t, l \rangle$ is assigned cost $l - \|f\|$.

When converting a digraph into an acyclic digraph by deleting vertices, an in-place conversion algorithm minimizes the amount of compression lost by selecting a set of vertices with the smallest total cost. This problem, called the FEEDBACK VERTEX SET problem, was shown by Karp [14] to be NP-hard for general digraphs. We have shown previously [7] that it remains NP-hard even when restricted to CRWI digraphs. Thus, we do not expect an efficient algorithm to minimize the cost in general.

For our implementation of in-place conversion, we examine two efficient, but not optimal, policies for breaking cycles. The *constant-time* policy picks the "easiest" vertex to remove, based on the execution order of the topological sort, and deletes this vertex. This policy performs no extra work when breaking cycles. The *local-minimum* policy detects a cycle and loops through all vertices in the cycle to determine and then delete the minimum cost vertex. The local-minimum policy may perform as much additional work as the total length of cycles found by the algorithm: $\mathrm{O}(n^2)$. Although these policies perform well in our experiments, we have shown previously [7] that they do not guarantee that the total cost of deletion is within a constant factor of the optimum.

4.4 Generating Conflict Free Permutations

Our algorithm for converting delta files into in-place reconstructible delta files takes the following steps to find and eliminate *WR* conflicts between a reference file and the new version to be materialized.

Algorithm

1. Given an input delta file, we partition the commands in the file into a set C of *copy* commands and a set A of *add* commands.
2. Sort the *copy* commands by increasing write offset, $C_{\text{sorted}} = \{c_1, c_2, ..., c_n\}$. For c_i and c_j, this set obeys: $i < j \longleftrightarrow t_i < t_j$. Sorting the copy commands allows us to perform binary search when looking for a copy command at a given write offset.
3. Construct a digraph from the *copy* commands. For the *copy* commands $c_1, c_2, ..., c_n$, we create a vertex set $V = \{v_1, v_2, ..., v_n\}$. Build the edge set E by adding an edge from vertex v_i to vertex v_j when *copy* command c_i reads from the interval to which c_j writes:

$$\overrightarrow{v_i v_j} \longleftrightarrow \textit{Read}_i \cap \textit{Write}_j \neq \emptyset \longleftrightarrow [f_i, f_i + l_i - 1] \cap [t_j, t_j + l_j - 1] \neq \emptyset.$$

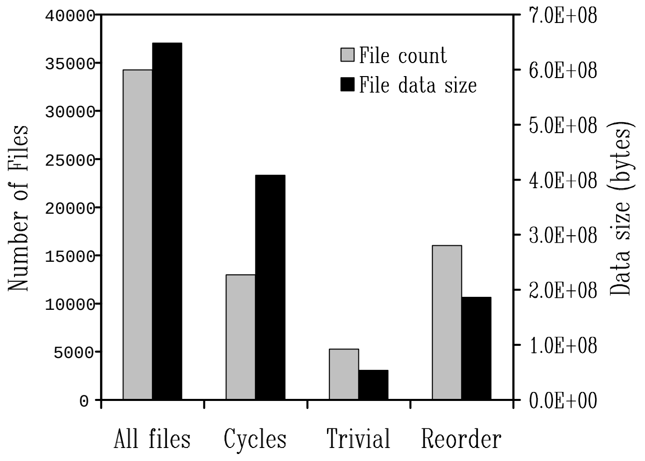


Figure 3: File counts and data size.

4. Perform a topological sort on the vertices of the digraph. This sort also detects cycles in the digraph and breaks them. When breaking a cycle, select one vertex on the cycle, using either the local-minimum or constant-time cycle breaking policy, and remove it. We replace the data encoded in its *copy* command with an equivalent *add* command, which is put into set A. The output of the topological sort orders the remaining *copy* commands so that they obey the property in Equation 2.
5. Output all *add* commands in the set A to the delta file.

The resulting delta file reconstructs the new version *out of order*, both out of write order in the version file and out of the order that the commands appeared in the original delta file.

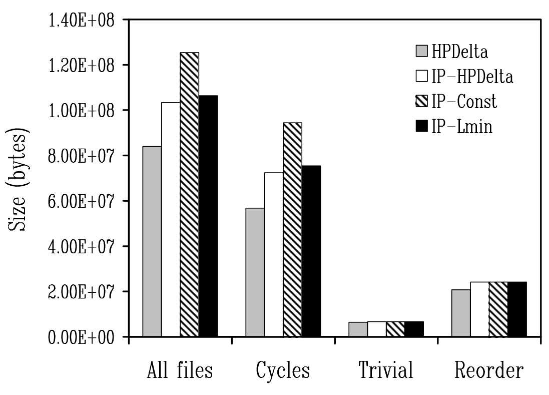
5 Experimental Results

As we are interested in using in-place reconstruction to distribute software, we extracted a large body of Internet available software and examined the compression and execution time performance of our algorithm on these files. Sample files include multiple versions of the GNU tools and the BSD operating system distributions, among other data, with both binary and source files being compressed and permuted for in-place reconstruction. These data were examined with the goals of:

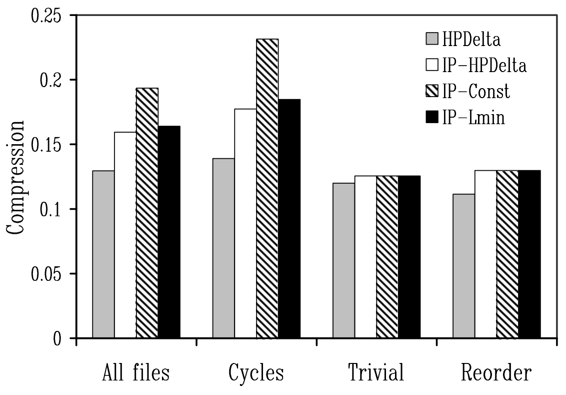
- determining the compression loss due to making delta files in-place reconstructible;
- comparing the the constant-time and local-minimum policies for breaking cycles;
- showing in-place conversion algorithms to be efficient when compared with delta compression algorithms on the same data; and
- characterizing the graphs created by the algorithm.

In all cases, we obtained the original delta files using the correcting 1.5-pass delta compression algorithm [3].

We categorize the delta files in our experiments into 3 groups that describe what operations were require to make files in-place reconstructible. Experiments were conducted over more than



(a) Delta size



(b) Delta compression

Figure 4: Compression performance

34,000 delta files totaling 6.5MB (Megabytes). Of these files (Figure 3), 63% of the files contained cycles that needed to be broken. 29% did not have cycles, but needed to have *copy* commands reordered. The remaining 8% of files were trivially in-place reconstructible; *i.e.,* none of the *copy* commands conflicted. For trivial files, performing copies before adds creates an in-place delta.

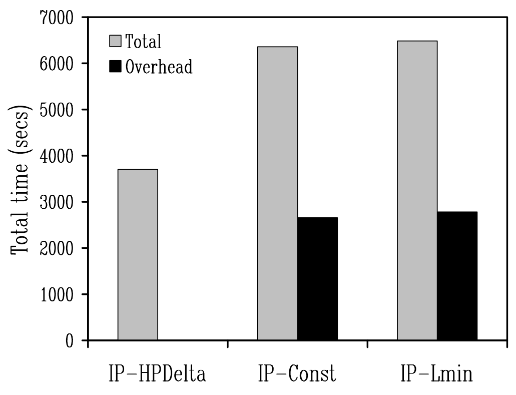
The amount of data in files is distributed differently across the three categories than are the file counts. Files with cycles contain over 4MB of data with an average file size of 31.4KB. Files that need copy commands reordered hold 1.9MB of data, with an average file size of 11.6KB. Trivially in-place reconstructible files occupy 585KB of data with an average file size of 10.2KB.

The distribution of files and data across the three categories confirms that efficient algorithms for cycle breaking and command reordering are needed to deliver delta compressed data in-place. While most delta files do not contain cycles, those that do have cycles contain the majority of the data.

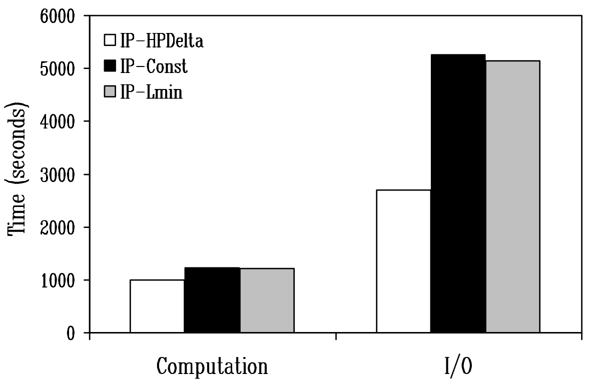
We group compression results into the same categories. Figure 4(a) shows the relative size of the delta files and Figure 4(b) shows compression (size of delta files as a fraction of the original file size). For each category and for all files, we report data for four algorithms: the unmodified correcting 1.5-pass delta compression algorithm [3] (HPDelta); the correcting 1.5-pass delta compression algorithm modified so that code-words are in-place reconstructible (IP-HPDelta); the in-place modification algorithm using the local-minimum cycle breaking policy (IP-Lmin); and the in-place modification algorithm using the constant-time cycle breaking policy (IP-Const).

The HPDelta algorithm is a linear time, constant space algorithm for generating delta compressed files. It outputs *copy* and *add* commands using a code-word format similar to industry standards [15].

The IP-HPDelta algorithm is a modification of HPDelta to output code-words that are suitable for in-place reconstruction. Throughout this paper, we have described *add* commands $\langle t, l \rangle$ and *copy* commands $\langle f, t, l \rangle$, where both commands encode explicitly the to t or write offset in the version file. However, delta algorithms that reconstruct data in write order need not explicitly encode a write offset – an *add* command can simply be $\langle l \rangle$ and a *copy* command $\langle f, l \rangle$. Since commands are applied in write order, the end offset of the previous command implies the write offset of the current command implicitly. The code-words of IP-HPDelta are modified to make the write offset explicit. The explicit write offset allows our algorithm to reorder copy commands. This extra field in each code-word introduces a per-command overhead in a delta file. The amount



(a) Overhead



(b) Breakdown

Figure 5: Run-time results

of overhead varies, depending upon the number of commands and the original size of the delta file. Encoding overhead incurs a 3% compression loss over all files.

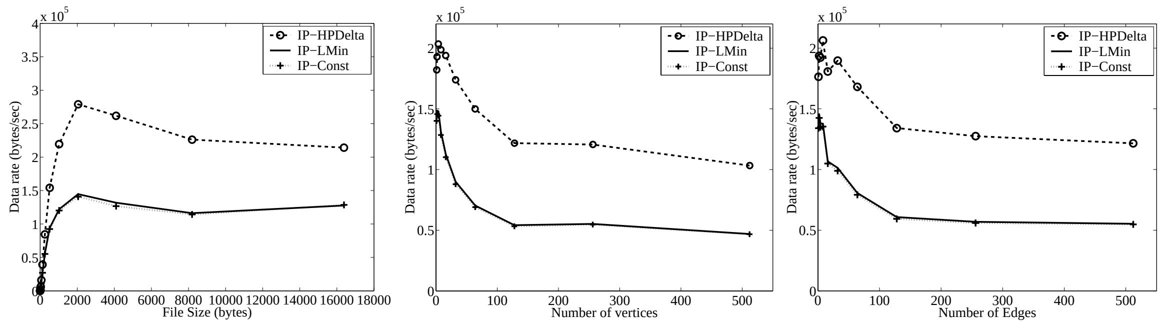
From the IP-HPDelta algorithm, we derive the IP-Const and IP-Lmin algorithms. They run the IP-HPDelta algorithm to generate a delta file and then permute and modify the commands according to our technique to make the delta file in-place reconstructible. The IP-Const algorithm implements the constant-time policy and the IP-Lmin algorithm implements the local-minimum policy.

Experimental results indicate the amount of compression lost due to in-place reconstruction and divides the loss into encoding overhead and cycle breaking. Over all files, HPDelta compresses data to 12.9% its original size. IP-HPDelta compresses data to 15.9%, losing 3% compression to encoding overhead. IP-Const loses an additional 3.4% compression by breaking cycles for a total compression loss of 6.4%. In contrast, IP-Lmin loses less than 0.5% compression for a total loss of less than 3.5%. The local-minimum cycle breaking policy performs excellently in practice, because compression losses are small when compared with encoding overheads. With IP-Lmin, cycle breaking accounts for less than 15% of the loss. IP-Const more than doubles the compression loss.

For reorder and trivial in-place delta files, no cycles are present and no compression lost. Encoding overhead makes up all lost compression – 0.5% for trivial delta files and 1.8% for reordered files.

Files with cycles exhibit an encoding overhead of 3.8% and lose 5.4% and 0.7% to cycle breaking for the IP-Const and IP-Lmin respectively. Because files with cycles contain the majority of the data, the results for files with cycles dominate the results for all files.

In-place algorithms incur execution time overheads when performing additional I/O and when permuting the commands in a delta file. An in-place algorithm must generate a delta file and then modify the file to have the in-place property. Since a delta file does not necessarily fit in memory, in-place algorithms create an intermediate file that contains the output of the delta compression algorithm. This intermediate output serves as the input for the algorithm that modifies/permutes commands. We present execution-time results in Figure 5(a) for both in-place algorithms – IP-Const and IP-Lmin. IP-Lmin and IP-Const perform all of the steps of the base algorithm (IP-HPDelta) before manipulating the intermediate file. Results show that the extra work incurs an



(a) File Size (b) Vertices (c) Edges

Figure 6: Run-time results

overhead of about 75%. However, figure 5(b) shows that almost all of this overhead comes from additional I/O. We conclude that the algorithmic tasks for in-place reconstruction are small when compared with the effort compressing data (about 10% the run-time) and miniscule compared to the costs of performing file I/O.

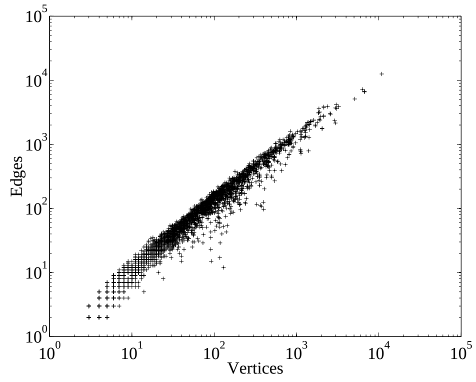
Despite inferior worst-case run-time bounds, the local-minimum cycle breaking policy runs faster than the constant-time policy in practice. Because file I/O dominates the run-time costs and because IP-Lmin creates a smaller delta file, it takes less total time than the theoretically superior IP-Const. In fact, IP-Const spends 2.2% more time performing I/O as a direct result of the files being 2.9% larger. IP-Lmin even uses slightly less time performing computation than IP-Const, which has to manipulate more data in memory.

Examining run-time results in more detail continues to show that IP-Lmin outperforms IP-Const, even for the largest and most complex input files. In Figure 6, we see how run-time performance varies with the input file size and with the size of the graph the algorithm creates (number of edges and vertices); these plots measure run time by data rate – file size (bytes) divided by run time (seconds).

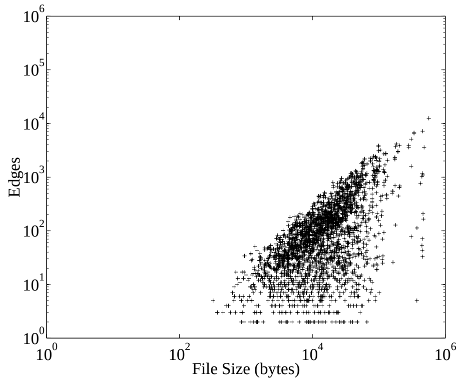
Owing to start-up costs, data rates increase with file size up to a point, past which rates tend to stabilize. The algorithms must load and initialize data structures. For small files, these costs dominate, and data rates are lower and increase linearly with the file size (Figure 6(a)). For files larger than 2000 bytes, rates tend to stabilize, exhibiting some variance, but neither increasing or decreasing as a trend. These results indicate that for inputs that amortize start-up costs, in-place algorithms exhibit a data rate that does not vary with the size of the input – a known property of the HPDelta algorithm [3]. IP-Lmin performs slightly better than IP-Const always.

The performance of all algorithms degrades as the size of the CRWI graphs increase. Figure 6(b) shows the relative performance of the algorithms as a function of the number of vertices, and Figure 6(c) shows this for the number of edges. For smaller graphs, performance degrades quickly as the graph size increases. For larger graphs, performance degrades more slowly. The graph size corresponds directly to the number of copy commands in a delta file. The more commands, the more I/O operations the algorithm must execute. Often more vertices means more small I/O rather than fewer large I/O, resulting in lower data rates.

Surprisingly, IP-Lmin continues to out-perform IP-Const even for the largest graphs. Analysis would indicate that the performance of IP-Lmin and IP-Const should diverge as the number of



(a) Edges versus Vertices



(b) Edges versus File Size

Figure 7: Edges in delta files that contain cycles.

edges increase. But no evidence of divergent performance exists. We attribute this to two factors: (1) graphs are relatively small and (2) all algorithms are I/O bound.

In Figure 7, we look at some statistical measures of graphs constructed when creating in-place delta files, restricted to those graphs that contain cycles. While graphs can be quite large, a maximum of 11503 vertices and 16694 edges, the number of edges scales linearly with the number of vertices and less than linearly with input file size. The constructed graphs do not exhibit edge relations that approach the $O(|V|^2)$ upper bound. Therefore, data rate performance should not degrade as the number of edges increases. For example consider two files as inputs to the IP-Lmin algorithm – one with a graph that contains twice the edges of the other. Based on our result, we expect the larger graph to have twice as many vertices and encode twice as much data. While the larger instance does twice the work breaking cycles, it benefits from reorganizing twice as much data, realizing the same data rate.

The linear scaling of edges with vertices and file size matches our intuition about the nature of delta compressed data. Delta compression encodes multiple versions of the same data. Therefore, we expect matching regions between these files (encoded as edges in a CRWI graph) to have spatial locality; *i.e.*, the same string often appears in the same portion of a file. These input data do not exhibit correlation between all regions of a file which would result in dense edge relations. Additionally, delta compression algorithms localize matching between files, correlating or synchronizing regions of file data [3]. All of these factors result in the linear scaling that we observe.

6 Conclusions

We have presented algorithms that modify delta files so that the encoded version may be reconstructed in the absence of scratch memory or storage space. Such an algorithm facilitates the distribution of software to network attached devices over low bandwidth channels. Delta compression lessens the time required to transmit files over a network by encoding the data to be transmitted compactly. In-place reconstruction exchanges a small amount of compression in order to do so without scratch space.

Experimental results indicate that converting a delta file into an in-place reconstructible delta file has limited impact on compression, less than 4% in total with the majority of compression loss from encoding overheads rather than modifications to the delta file. We also find that for bottom line performance keeping delta files small to reduce I/O matters more than execution time differences in cycles breaking heuristics, because in-place reconstruction is I/O bound. For overall performance, the algorithm to convert a delta file to an in-place reconstructible delta file requires less time than generating the delta file in the first place.

In-place reconstructible delta file compression provides the benefits of delta compression for data distribution to an important class of applications – devices with limited storage and memory. In the current network computing environment, this technology decreases greatly the time to distribute content without increasing the development cost or complexity of the receiving devices. Delta compression provides Internet-scale file sharing with improved version management and update propagation, and in-place reconstruction delivers the technology to the resource constrained computers that need it most.

7 Future Directions

Detecting and breaking conflicts at a finer granularity can reduce lost compression when breaking cycles. In our current algorithms, we eliminate cycles by converting *copy* commands into *add* commands. However, typically only a portion of the offending *copy* command actually conflicts with another command; only the overlapping range of bytes. We propose, as a simple extension, to break a cycle by converting part of a *copy* command to an *add* command, eliminating the graph edge (rather than a whole vertex as we do today), and leaving the remaining portion of the *copy* command (and its vertex) in the graph. This extension does not fundamentally change any of our algorithms, only the cost function for cycle breaking.

As a more radical departure from our current model, we are exploring reconstructing delta files with bounded scratch space, as opposed to zero scratch space as with in-place reconstruction. This formulation, suggested by Martín Abadi, allows an algorithm to avoid *WR* conflicts by moving regions of the reference file into a fixed size buffer, which preserves reference file data after that region has been written. The technique avoids compression loss by resolving data conflicts without eliminating *copy* commands.

Reconstruction in bounded space is logical, as target devices often have a small amount of available space that can be used advantageously. However, in-place reconstruction is more generally applicable. For bounded space reconstruction, the target device must contain enough space to rebuild the file. Equivalently, an algorithm constructs a delta file for a specific space bound. Systems benefit from using the same delta file to update software on many devices. For example, distributing an updated product list to many PDAs in the same sales force. In such cases, in-place reconstruction offers a lowest common denominator solution in exchange for a little lost compression.

We also are developing algorithms that can perform peer-to-peer style delta compression [26] in an in-place fashion. This allows delta compression to be used between two versions of a file stored on separate machines and is often a more natural formulation, because it does not require a computer to maintain the original version of data to employ delta compression. This works well for file systems, most of which do not handle multiple versions.

Our ultimate goal is to use in-place algorithms as a basis for a data distribution system. The system will operate both in hierarchical (client/server) and peer-to-peer modes. It will also conform to Internet standards [15] and, therefore, work seamlessly with future versions of HTTP.

References

[1] The free network project – rewiring the Internet. Technical Report http://freenet.sourceforge.net/, 2001.

[2] The gnutella protocol specification. Technical Report http://www.gnutelladev.com/protocol/gnutella-protocol.html, 2001.

[3] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. `www.almaden.ibm.com/cs/people/stock/diff7.ps`, IBM Research Report RJ 10187, April 2000 (revised Aug. 2001).

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic deltas for WWW latency reduction. In *Proceedings of the 1998 Usenix Technical Conference*, 1998.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison–Wesley Publishing Co., 1987.

[6] R. C. Burns and D. D. E. Long. Efficient distributed backup and restore with delta compression. In *Proceedings of the Fifth Workship on I/O in Parallel and Distributed Systems, San Jose, CA*, November 1997.

[7] R. C. Burns and D. D. E. Long. In-place reconstruction of delta compressed files. In *Proceedings of the Seventeenth ACM Symposium on Principles of Distributed Computing*, 1998.

[8] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE Infocom '99 Conference, New York, NY*, March 1999.

[9] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, May 1997.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[11] S. P. de Jong. Combining of changes to a source file. *IBM Technical Disclosure Bulletin*, 15(4):1186–1188, September 1972.

[12] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Proceedings of the 6th Workshop on Software Configuration Management*, March 1996.

[13] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[14] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[15] D. G. Korn and K.-P. Vo. The VCDIFF generic differencing and compression format. Technical Report Internet-Draft draft-vo-vcdiff-00, Internet Engineering Task Force (IETF), 1999.

[16] S. Kurtz. Reducing the space requirements of suffix trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[17] J. P. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. In *Proceedings System Configuration Management*, 1998.

[18] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2), April 1978.

[19] W. Miller and E. W. Myers. A file comparison program. *Software – Practice and Experience*, 15(11):1025–1040, November 1985.

[20] J. C. Mogul, F. Douglis, A. Feldman, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of ACM SIGCOMM '97*, September 1997.

[21] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management, Trondheim, Norway, 12-14 June 1991*, pages 144–152. ACM, June 1991.

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

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

[24] W. F. Tichy. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems*, 2(4), November 1984.

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

[26] A. Tridgell and P. Mackerras. The RSync algorithm. Technical Report TR-CS-96-05, The Australian National University, 1996.

[27] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.