A LINEAR TIME, CONSTANT SPACE DIFFERENCING ALGORITHM

Randal C. Burns and Darrell D. E. Long

Department of Computer Science
University of California Santa Cruz
Santa Cruz, California 95064

ABSTRACT

An efficient differencing algorithm can be used to compress version of files for both transmission over low bandwidth channels and compact storage. This can greatly reduce network traffic and execution time for distributed applications which include software distribution, source code control, file system replication, and data backup and restore.

An algorithm for such applications needs to be both general and efficient; able to compress binary inputs in linear time. We present such an algorithm for differencing files at the granularity of a byte. The algorithm uses constant memory and handles arbitrarily large input files. While the algorithm makes minor sacrifices in compression to attain linear runtime performance, it outperforms the byte-wise differencing algorithms that we have encountered in the literature on all inputs.

I. INTRODUCTION

Differencing algorithms compress data by taking advantage of statistical correlations between different versions of the same data sets. Strictly speaking, they achieve compression by finding common sequences between two versions of the same data that can be encoded using a copy reference.

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

$$F_{\text{base}} + F_{\text{version}} \rightarrow \Delta_{(\text{base,version})} \quad (1)$$

Reconstruction, the inverse operation, requires the base file and a delta file to rebuild a version.

$$F_{\text{base}} + \Delta_{(\text{base,version})} \rightarrow F_{\text{version}} \quad (2)$$

One encoding of a delta file consists of a linear array of editing directives (Figure 1). These directives are copy commands, references to a location in a base file where the same data exists, and add commands, instructions to add data into the version file followed by the data to be added. While there are other representations [12, 1, 3], in any encoding scheme, a differencing algorithm must have found the copies and adds to be encoded. So, any encoding technique is compatible with the methods that we present.

Several potential applications of version differencing motivate the need for a compact and efficient differencing algorithm. An efficient algorithm could be used to distribute software over a low bandwidth network such as a modem or the Internet. Upon releasing a new version of software, the version could be differenced with respect to previous version. With compact versions, a low bandwidth channel can effectively distribute a new release of dynamically self updating software in the form of a binary patch. This technology has the potential to greatly reduce time to market on a new version and ease the distribution of software customizations.

For replication in distributed file systems, differencing can reduce by a large factor the amount of information that needs to be updated by transmitting deltas for all of the modified files in the replicated file set.

In distributed file system backup and restore, differential compression would reduce the time to perform file system backup, decrease network traffic during backup and restore, and lessen the storage to maintain a backup image [7]. Backup and restore can be limited by both bandwidth on the network, often 10 MB/s, and poor throughput to secondary and tertiary storage devices, often 500 KB/s to tape storage. Since resource limitations frequently make backing up the just the changes to a file system infeasible over a single night or even weekend, differential file compression has great potential to alleviate bandwidth problems by using available processor cycles to reduce the amount of data transferred. This technology can be used to provide backup and restore services on a subscription basis over any network including the Internet.

Differencing has it origins in both longest common subsequence (LCS) algorithms and the string-to-string correction problem [13]. Some of the first applications of differencing updated the screens of slow terminals by sending a set of edits to be applied locally rather than retransmitting a screen full of data. Another early application was the UNIX diff utility which used the LCS method to find and output the changes to a text file. diff was useful for source code development and primitive document control.

0-7803-3873-1/97 $10.00 © 1997 IEEE

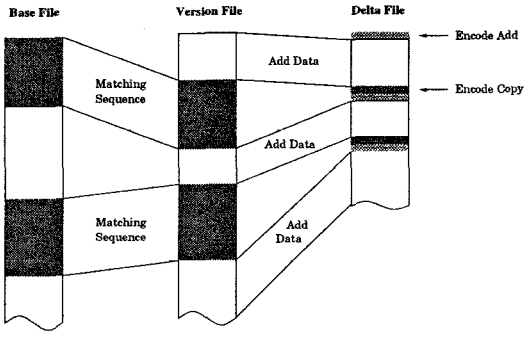


Figure 1: The copies found between the base and version file are encoded as copy commands in the delta file. Unmatched sequences are encoded as an add command followed by the data to be added.

LCS algorithms find the longest common sequence between two strings by optimally removing symbols in both files leaving identical and sequential symbols.[1] While the LCS indicates the sequential commonality between strings, it does not necessarily detect the minimum set of changes. More generally, it has been asserted that string metrics that examine symbols sequentially fail to emphasize the global similarity of two strings [4]. Miller and Myers [6] established the limitations of LCS when they produced a new file compare program that executed at four times the speed of the diff program while producing significantly smaller deltas.

The *edit distance* [10] proved to be a better metric for the difference of files and techniques based on this method enhanced the utility and speed of file differencing. The edit distance assigns a cost to edit operations such as delete a symbol, insert a symbol, and copy a symbol. For example, the LCS between strings ***xyz*** and ***xzy*** is ***xy***, which neglects the common symbol ***z***. Using the edit distance metric, ***z*** may be copied between the two files producing a smaller change cost than LCS. In the string-to-string correction problem [13], an algorithm minimizes the edit distance to minimize the cost of a given string transformation.

Tichy [10] adapted the string-to-string correction problem to file differencing using the concept of block move. Block move allows an algorithm to copy a string of symbols rather than an individual symbol. He then applied the algorithm to source code revision control package and created RCS [11]. RCS detects the modified lines in a file and encodes a delta file by adding these lines and indicating lines to be copied from the base version of a file. We term this differencing at *line granularity*. The delta file is a line by line edit script applied to a base file to convert it to the new version. Although the SCCS version control system [9] precedes RCS, RCS generates "minimal" line granularity delta files and is the definitive previous work in version control.

While line granularity may seem appropriate for source code, the concept of revision control needs to be generalized to include binary files. This allows data, such as edited multimedia, to be revised with the same version control and recoverability guarantees as text. Whereas revision control is currently a programmers tool, binary revision control systems will enable the publisher, film maker, and graphic artist to realize the benefits of strict versioning. It also enables developers to place bitmap data, resource files, databases and binaries under their revision control system. Some previous packages have been modified to handle binary files, but in doing so they imposed an arbitrary line structure. This results in delta files that achieve little or no compression as compared to storing the versions uncompressed.

Recently, an algorithm appeared that addresses differential compression of arbitrary byte streams [8]. The algorithm modifies the work of Tichy [10] to work on bytewise data streams rather than line oriented data. This algorithm adequately manages binary sources and is an effective developer's tool for source code control. However, the algorithm exhibits execution time quadratic in the size of the input, $O(M \times N)$ for files of size M and N. The algorithm also uses memory linearly proportional to the size of the input files, $O(M + N)$. To find matches the algorithm implements the greedy method, which we will show to be optimal under certain constraints. The algorithm will then be used as a basis for comparison.

As we are interested in applications that operate on all data in a network file system, quadratic execution time renders differencing prohibitively expensive. While it is a well known result that the majority of the files are small, less than 1 kilobyte [2], a file system has a minority of files that may be large, ten to hundreds of megabytes. In order to address the differential compression of large files, we devised an differencing algorithm that runs in both linear time, $O(M + N)$, and constant space, $O(1)$.

Section II outlines the greedy differencing algorithm, proves it optimal, and establishes that the algorithm takes quadratic execution time. Section III presents the linear time differencing algorithm. Section IV analyzes the linear time algorithm for run-time and compression performance. Section V presents an experimental comparison of the linear time algorithm and the greedy algorithm. We conclude in Section VI that the linear time algorithm provides near optimal compression and the efficient performance required for distributed applications.

II. GREEDY METHODS FOR FILE DIFFERENCING

Greedy algorithms often provide simple solutions to optimization problems by making what appears to be the best decision, the greedy decision, at each step. For differencing files, a greedy algorithm takes the longest match it can find at a given offset on the assumption that this match

[1] A string/substring contains all consecutive symbols between and including its first and last symbol whereas a sequence/subsequence may omit symbols with respect to the corresponding string.

provides the best compression. Greedy makes a locally optimal decision with the hope that this decision is part of the optimal solution over the input.

For file differencing, we prove the greedy algorithm provides an optimal encoding of a delta file and show that the greedy technique requires time proportional to the product of the sizes of the input files. Then we present an algorithm which approximates the greedy algorithm in linear time and constant space by finding the match that appears to be the longest without performing exhaustive search for all matching strings.

A. Examining Greedy Delta Compression

For our analysis, we consider delta files constructed by a series of editing directives; "add commands", followed by the data to be added, and "copy commands" that copy data from the base file into the version file using an offset and length (Figure 1).

Given a base file and another version of that base file, the greedy algorithm for constructing differential files finds the longest copy in the base file from the first offset in the version. It then looks for the longest copy starting at the next offset. If at a given offset, it cannot find a copy, the symbol at this offset is marked to be added and the algorithm advances to the following offset. For an example of a greedy differencing algorithm refer to the work of Reichenberger [8].

We now prove that the greedy algorithm is optimal for a simplified file encoding scheme. In this case an optimal algorithm produces the smallest output delta. For binary differencing, symbols in the file may be considered bytes and a file a stream of symbols. However, this proof applies to differencing at any granularity. We introduce and use the concept *cost* to mean the length (in bits) for the given encoding of a string of symbols.

Claim Given a base file B, a version of that base file V, and an alphabet of the symbols Σ, by making the following assumptions:

- A copy of any length may be encoded with a unit cost = c.
- All symbols in the alphabet Σ appear in the base file B.
- Copying a string of length l with maximum cost $c \times l$ provides an encoding as compact as adding the same string.

we can state:

Theorem 1 *The greedy algorithm finds an optimal encoding of the version file V with respect to the base file B.*

Proof Since all symbols in the alphabet Σ appear in the base file B, a symbol or string of symbols in the version file V may be represented in a differential file D exclusively by a copy or series of copies from B. Since we have assumed a unit cost function for encoding all copies and this cost is less than or equal to the cost of adding a symbol in the version file, there exists an optimal representation P, of V with respect to B, which only copies strings of symbols from B. In order to prove the optimality of a greedy encoding G, we require the intermediate result of Lemma 1.

Lemma 1 *For an arbitrary number of copies encoded, the length of version file data encoded by the greedy encoding is greater than or equal to the length of data encoded by optimal encoding.*

Proof *(by induction)* We introduce p_i to be the length of the i^{th} copy in the optimal encoding P and g_i to be the length of the i^{th} copy in the greedy encoding G. The length of data encoded in P and G after n copies are respectively given by:

$$\sum_{i=1}^{n} p_i \text{ and } \sum_{i=1}^{n} g_i$$

1. At file offset 0 in V, P has a copy command of length p_1. G encodes a matching string of length g_1 which is the longest string starting at offset 0 in V. Since G encodes the longest possible copy, $g_1 \geq p_1$.
2. Given that G and P have encoded $n-1$ copies and the current offset in G is greater than the current offset in P, we can conclude that after G and P encode an n^{th} copy that the offset in G for n copies is greater than the offset in P.

$$\sum_{i=1}^{n-1} g_i \geq \sum_{i=1}^{n-1} p_i \implies \sum_{i=1}^{n} g_i \geq \sum_{i=1}^{n} p_i \quad (3)$$

G encodes a copy of length g_n and P encodes a copy of length p_n. If equation 3 did not hold, P would have found a copy of length p_n at offset $\sum_{i=1}^{n-1} p_i$ that is greater than $g_n + \sum_{i=1}^{n-1} g_i - \sum_{i=1}^{n-1} p_i$. A substring of this copy would be a string starting at $\sum_{i=1}^{n-1} g_i$ of length greater than g_n. As G always encodes the longest matching string, in this case g_n, this is a contradiction and equation 3 must hold. ■

Having established Lemma 1, we conclude that the number of copy commands that G uses to encode V is less than or equal to the number of copies used by P. However, since P is an optimal encoding, the number of copies P uses to encode V is less than or equal to the number the G uses. We can therefore state that, size(G) = size(P) = $c \times N$ where N is the number of copy commands in greedy encoding. ■

We have shown that the greedy algorithm provides an optimal encoding of a version file. Practical elements of

the algorithm weaken our assumptions. Yet, the greedy algorithm consistently reduces files to near optimal and should be considered a minimal differencing algorithm.

B. Analysis of Greedy Methods

Common strings may be quickly identified as they also have common *footprints*. In this case a *footprint* is the value of a hash function over a fixed length prefix of a string. The greedy algorithm must examine all matching footprints and extend the matches in order to find the longest matching string. The number of matching footprints between the base and version file can grow with respect to the product of the sizes of the input files, *i.e.* $O(M \times N)$ for files of size M and N, and the algorithm uses time proportional to this value.

In practice, many files elicit this worst case behavior. In both database files and executable files, binary zeros are stuffed into the file for alignment. This "zero stuffing" creates frequently occuring footprints which must all be examined by the algorithm.

Having found a footprint in the version file, the greedy algorithm must compare this footprint to all matching footprints in the base file. This requires it to maintain a canonical listing of all footprints in one file, generally kept by computing and storing a hash value over all string prefixes [8]. Consequently, the algorithm uses memory proportional to the size of the input, $O(N)$, for a size N base file.

III. VERSIONING IN LINEAR TIME

Having motivated the need to difference all files in a file system and understanding that not all file are small [2], we improve upon both the runtime performance bound and runtime memory utilization of the greedy algorithm. Our algorithm intends to find matches in a greedy fashion but does not guarantee to execute greedy exactly.

A. A Linear Time Differencing Algorithm

The linear algorithm modifies the greedy algorithm in that it attempts to take the longest match at a given offset by taking the longest matching string at the first matching string prefix beyond the offset at which a previous match was encoded; we term this the *next match* policy. In many instances matching strings are sequential between file versions, *i.e.* they occur in the same order in both files. When strings that match are sequential, the next matching prefix approximates the best match extremely well. In fact this property holds for all changes that are insertions and deletions (Figure 2). We expect many files to exhibit this property, most notably mail, database, image and log files.

The linear time differencing algorithm takes as input two files, usually versions of each other, and using one hash table performs the following actions:

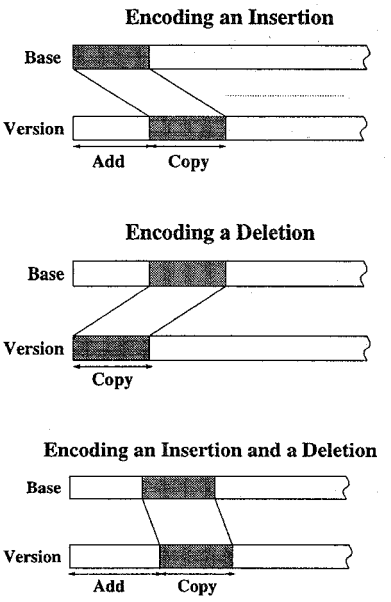


Figure 2: Simple file edits consist of insertions, deletions and combinations of both. The linear time algorithm finds and encodes all modifications that meet the simple edit criteria.

Algorithm

1. Start file pointers b_{offset} in the base file and v_{offset} in the version file at file offset zero. Create a footprint for each offset by hashing a prefix of bytes. Store the start position in the version file as v_{start}.
2. We call this state "hashing mode". For each footprint:
   (a) If there is no entry in the table at that footprint value, make an entry in the hash table. An entry will indicate the file and offset from which the footprint was hashed.
   (b) If there is an entry at a footprint value, if the entry is from the other version of the file, verify that the prefixes are identical. If the prefixes prove to be the same, matching strings have been found. Continue with step 3.
   (c) If there is an entry at a footprint value and the entry is from the same file, retain the existing hash entry.

   Advance both v_{offset} and b_{offset} one byte, hash prefixes, and repeat step 2.
3. Having found a match at step 2b, leave hashing mode and enter "identity mode". Given matching prefixes between some offset v_{copy} in the version file, and some offset b_{copy} in the base file, match bytes forward in the files to find the longest match of length l. Set v_{offset} and b_{offset} to the ends of the match.
4. Encode the region of the version file from v_{start} to v_{copy} using an add codeword followed by the data to be added. Encode the region from v_{copy} to v_{offset} in the version file using a copy codeword encoding l, the length of the copy found, and b_{copy}, the offset in the base file.

5. Flush the hash table to remove the information about the files previous to this point. Set v_{start} to the v_{offset} and repeat step 2.

By flushing the hash table, the algorithm enforces the next match policy. Note that a match can be between the current offset in one version of the file and a previous offset in the other version. After flushing the hash table, the algorithm effectively remembers the first instance of every footprint that it has seen since encoding the last copy.

IV. ANALYSIS OF THE LINEAR TIME ALGORITHM

We often expect the changes between two versions of a file to be simple edits, insertions of information and deletions of information. This property implies that the common strings that occur in these files are sequential. An algorithm can then find all matching strings in a single pass over the inputs files. After finding a match, we can limit our search space for subsequent matches to only the file offsets greater than the end of the previous matching string.

Many files exhibit insert and delete only modifications. In particular mail files and database files. Mail files have messages deleted out from the middle of the file and data appended to the end. Relational database files operate on tables of records, appending records to the end of a table, modifying records in place, and deleting them from the middle of the table. System logs have an even more rigid format as they are append only files.

When a match is found and the algorithm enters identity mode, if the match is not spurious (section B), the pointers are "synchronized", indicating that the current offset in the version file represents the same data at the offset in the base file. The algorithm's two phases, hashing and identity, represent the synchronization of file offsets and copying from synchronized offsets. When the identity test fails, the files differ and the file offsets are again "out of synch". Then, the algorithm enters hashing mode to regain the common location of data in the two files.

We selected the Karp–Rabin hashing function [5] for generating footprints as it can be calculated incrementally, *i.e.* a footprint may be evaluated from the footprint at the previous offset and the last byte of the current string prefix. This technique requires fewer operations when calculating the value of overlapping footprints sequentially. Our algorithm always hashes successive offsets in hashing mode and realizes significant performance gains when using this function.

A. Performance Analysis

The presented algorithm operates both in linear time and constant space. At all times, the algorithm maintains a hash table of constant size. After finding a match, hash entries are flushed and the same hash table is reused to find the next matching prefix. Since this hash table neither grows nor is deallocated, the algorithm operates in constant space, roughly the size of the hash table, on all inputs.

Since the maximum number of hash entries does not necessarily depend on the file input size, the size of the hash table need not grow with the size of the file. The maximum number of hash entries is bounded by twice the number of bytes between the end of the previous copied string and the following matching prefix. On highly correlated files, we would expect a small maximum number of hash entries since we expect to find matching strings frequently.

The algorithm operates in time linear in the size of the input files as we are guaranteed to advance either the base file offset or the version file offset by one byte each time through the inside loop of the program. In identity mode, both the base file offset and the version file offset are incremented by one byte at each step. Whereas in hashing mode, each time a new offset is hashed, at least one of the offsets is incremented, as matching prefixes are always found between the current offset in one file and a previous offset in another. Therefore, identity mode proceeds through the input at as much as twice the rate of hashing mode. Furthermore, the byte identity function is far easier to compute than the Karp–Rabin [5] hashing function. On highly correlated files, we expect the algorithm to spend more time in identity mode than it would on less correlated versions. We can then state that the algorithm executes faster on more highly correlated inputs and the linear algorithm operates best on its most common input, similar version files.

B. Sub–optimal Compression

The algorithm achieves less than optimal compression when either the algorithm falsely believes that the offsets are synchronized, the assumption that all changes between versions consist of insertions and deletions fails to hold, or when the implemented hashing function exhibits less than ideal behavior.

Due to the assumption of changes being only inserts and deletes, the algorithm fails to find rearranged strings. Upon encountering a rearranged string, the algorithm takes the next match it can find. This leaves some string in either the base file or in the version file that could be compressed and encoded as a copy, but will be encoded as an add, achieving no additional compression. In Figure 3, the algorithm fails to find the copy of tokens ABCD since the string has been rearranged. In this simplified example we have selected a prefix of length one. The algorithm encodes EFG as a copy and flushes the hash table, removing symbols ABCD that previously appeared in the base file. When hashing mode restarts the match has been missed and will be encoded as an add.

The algorithm is also susceptible to spurious hash collisions as a result of taking the next match rather than the

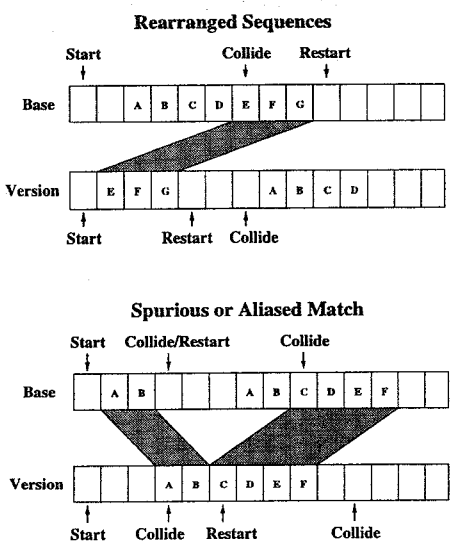


Figure 3: Sub-optimal compression may be achieved due to the occurrence of spurious matches or rearranged strings. The encoded matches are shaded.

best match. These collisions indicate that the algorithm believes that it has found synchronized offsets between the files when in actuality the collision just happens to be between strings that match by chance. In Figure 3, the algorithm misses the true start of the string ABCDEF in the base file (*best match*) in favor of the previous string at AB (*next match*). Upon detecting and encoding a "spurious" match, the algorithm achieves some degree of compression, just not the best compression. Furthermore, the algorithm never bypasses "synchronized offsets" in favor of a spurious match. This also follow directly from choosing the next match and not the best match. This result may be generalized. Given an ideal hash function, the algorithm never advances the file offsets past a point of synchronization.

Hashing functions are, unfortunately, not ideal. Consequently, the algorithm may also experience the *blocking* of footprints. For a new footprint, if there is another footprint from the same file already occupying that entry in the hash table, the second footprint is ignored and the first one retained. In this instance, we term the second footprint to be *blocked*. This is the correct procedure to implement the next match policy assuming that all footprints represent a unique string. However, hash functions generally hash a large number of inputs to a smaller number of keys and are therefore not unique. Strings that hash to the same value may differ and the algorithm loses the ability to find strings matching the discarded string prefix.

Footprint blocking could be addressed by any rehash function or hash chaining. However, this solution would destroy the constant space utilization bound on the algorithm. Instead of a rehash function, we propose to address footprint blocking by scanning both forwards and backwards in identity mode. This simple modification allows the algorithm to go back and find matches starting at a prefix that was hash blocked. The longer the matching string, the less likely that match will be blocked as this requires consecutive blocked prefixes. Under this solution, the algorithm still operates in constant space, and although matches may still be blocked, the probability of blocking a match decreases geometrically with the length of the match.

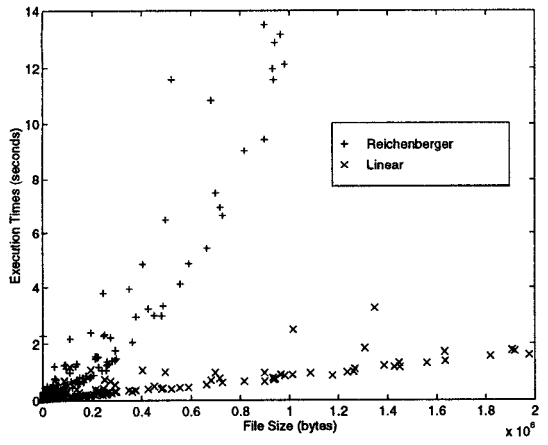
V. EXPERIMENTAL RESULTS

We compared the Reichenberger [8] greedy algorithm against our linear time algorithm to experimentally verify the performance improvements and quantify the amount of compression realized. The algorithms were run against multiple types of data that are of interest to potential applications. Data include mail files, modified and recompiled binaries, and database files.

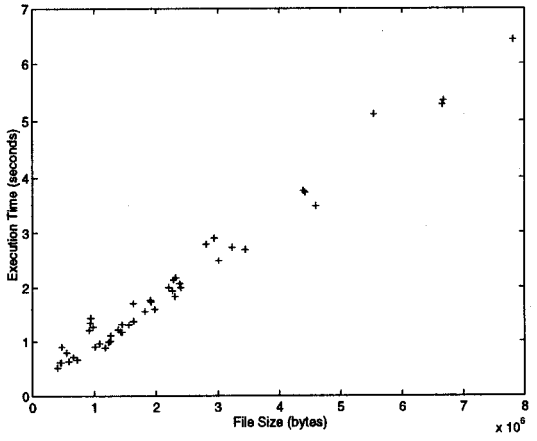
Both algorithms, where appropriate, were implemented with the same facilities. This includes the use of the Reichenberger codewords for encoding copy and add commands in the delta file, memory mapped I/O, the use of the same prefix length for footprint generation, and the use of the Karp–Rabin hashing algorithm in both cases. Karp–Rabin hashing is used by the Reichenberger algorithm since it also realizes benefits from incremental hashing, by sequentially hashing one whole file before searching for common strings.

The linear algorithm outperforms the Reichenberger algorithm on all inputs, operating equally as fast on very small inputs and showing significant performance gains on all inputs larger than a few kilobytes. The performance curve of the Reichenberger algorithm grows quadratically with the file input size. The algorithm consistently took more than 10 seconds to difference a 1MB file, extrapolating this curve to a 10MB file, the algorithm would complete in slightly more than 15 minutes. Depending upon the machine, the linear algorithm can compress as much as several megabytes per second. Currently, the data throughput is I/O bound when performing the byte identity function and processor bound when performing the hashing function. The relative data rates are approximately 10 MB/s and 280 KB/s for identity and hashing mode respectively. These results were attained upon an IBM 43P PowerPC with a 133MHz processor and a local F/W SCSI hard drive on a 10 MB/s data bus.

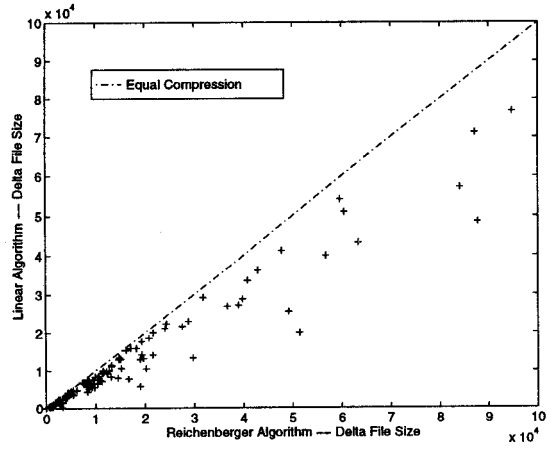
In Figure 4a, the runtime performance of the Reichenberger algorithm grows in a quadratic fashion, whereas the linear time algorithm exhibits growth proportional to the file input size. We also show that our algorithm's execution time continues to grow linearly on large input files in Figure 4b. There is a high amount of variability in the time performance of the linear algorithm on a file of any given size depending upon how long the algorithm spends in hashing mode as compared to identity mode.



(a) A comparison of the algorithms' execution time performance on inputs of less than 2MB.



(b) The linear algorithm's execution time performance on inputs as large as 8MB.



(c) Relative compression of linear algorithm with respect to the Reichenberger algorithm for delta files less than 100KB.

Figure 4: Experimental Results

Data	Linear Algorithm			Reichenberger Algorithm		
	Total	Mean	Std. Dev.	Total	Mean	Std. Dev.
Mail 1	11.5%	27.1%	38.1%	9.3%	19.9%	29.8%
Mail 2	3.2%	5.7%	7.2%	*	*	*
DB 1	0.7%	5.3%	10.9%	0.4%	3.9%	5.4%
DB 2	23.5%	17.5%	25.3%	17.4%	11.8%	15.2%
Binary	36.8%	33.4%	28.3%	31.5%	28.0%	24.9%

Table 1: The relative size of the delta files measured in the percent size as compared to the version file. Total is the compression over the sum of all files. The "*" indicates a data set of files too large for the Reichenberger algorithm.

Most of the data that we experimented on shows a high degree of compressibility, with some instances of databases showing the best compressibility. Our data sets (table 1) include: **Mail 1**, electronic mail files less than 1MB from a UNIX network file system; **Mail 2**, files greater than 1MB from the same system; **DB 1**, the weekly backup of a university student information database; **DB 2**, the same data from a different pair of weeks; and the **Binary** entry represents the compressibility of for all executable versions of both algorithms over their development cycle. The total compressibility represents the ratio between the sum of the sizes of the delta files compared to the sum of the sizes of the version files. Generally, the mean file size is larger than the total indicating that larger files in any data set tend to be more compressible. The standard deviation indicates the volatility of the data set with a high deviation showing data with more files that have been significantly altered.

Compressibility figures depend totally on the input data and are not meant to indicate that delta file compression achieves these kinds of results on all inputs. Rather, the data is, in our experience, representative of the compression that can be achieved on versions in a typical UNIX file system. We consistently noted that the data sets with larger files also tended to be more compressible. This is verified by the data in table 1. **Mail 2** consists of files larger than 1MB and are 10% more compressible than the files in **Mail 1**, files less than 1MB.

The linear algorithm consistently compressed data to within a small factor of the compression the greedy algorithm realizes. On all the mail files less than 1MB, the linear algorithm achieved compression to less than 12% the original size whereas the Reichenberger algorithm compressed the files less than 3% more to slightly under 9%. The relative compression of the algorithms are displayed in Figure 4c. Points on the unit slope line represent files that were compressed equally by both methods. The Reichenberger encoding is consistently equal to or more compact than the linear algorithm, but only by a small factor.

Experimental results indicate the suitability of our methods to many applications as compression by a factor of 30 or more is feasible on many data sets. The results also indicate that the linear algorithm consistently performs well in compressing versions when compared with the greedy algorithm. The linear algorithm provides near optimal compression and does so in linear time.

VI. SUMMARY AND CONCLUSIONS

We have described a differencing algorithm that executes in both linear time and constant space. This algorithm executes significantly faster than the greedy algorithm and provides comparative compression to the greedy method, which has been shown to provide optimal compression. The linear algorithm approximates the greedy algorithm by taking the next matching string following the previous match, rather than exhaustively searching for the best match over the whole file. This next match policy corresponds highly with best match when files are versions with insert and delete modifications. The algorithm enforces the next match policy by synchronizing pointers between two versions of a file to locate similar data.

Experiments have shown the linear time algorithm to consistently compress data to within a small percentage of the greedy algorithm and to execute significantly faster on inputs of all sizes. Results have also shown many types of data to exhibit high correlation among versions and differencing can efficiently compress the representation of these files.

We envision a scalable differencing algorithm as an enabling technology that permits files of any size and format to be placed under version control, and allows the transmission of new version of files over low bandwidth channels. File differencing can mitigate the transmission time and network traffic for any application that manages distributed views of changing data. This includes replicated file systems and distributed backup and restore. A technology that was previous relegated to source code control may be generalized with this algorithm and applied to address network resource limitations for distributed applications.

VII. ACKNOWLEDGMENTS

We wish to thank and credit Dr. Robert Morris of the IBM Almaden Research Center for his innovation in the application of delta compression. We also wish to thank Professor David Helmbold and the research group of Dr. Ronald Fagin for their assistance in verifying and revising our methods, Mr. Norm Pass for his support of this effort, and the University of California Santa Cruz which provided us with file system data.

VIII. REFERENCES

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

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

[3] BLACK, A. P., AND CHARLES H. BURRIS, JR. A compact representation for file versions: A preliminary report. In *Proceedings of the 5th International Conference on Data Engineering* (1989), IEEE, pp. 321–329.

[4] EHRENFEUCHT, A., AND HAUSSLER, D. A new distance metric on string computable in linear time. *Discrete Applied Mathematics 20* (1988), 191–203.

[5] KARP, R. M., AND RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development 31*, 2 (1987), 249–260.

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

[7] MORRIS, R. Conversations regarding differential compression for file system backup and restore, Feb. 1996.

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

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

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

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

[12] TSOTRAS, V., AND GOPINATH, B. Optimal versioning of objects. In *Proceeedings of the Eight International Conference on Data Engineering, Tempe, AZ, USA, 2-3 Feb. 1992* (Feb. 1992), IEEE, pp. 358–365.

[13] WAGNER, R., AND FISCHER, M. The string-to-string correction problem. *Journal of the ACM 21*, 1 (Jan. 1973), 168–173.