An approximation to the greedy algorithm for differential compression

R. C. Agarwal
K. Gupta
S. Jain
S. Amalapurapu

We present a new differential compression algorithm that combines the hash value techniques and suffix array techniques of previous work. The term "differential compression" refers to encoding a file (a version file) as a set of changes with respect to another file (a reference file). Previous differential compression algorithms can be shown empirically to run in linear time, but they have certain drawbacks; namely, they do not find the best matches for every offset of the version file. Our algorithm, ***hsadelta*** *(hash suffix array delta), finds the best matches for every offset of the version file, with respect to a certain granularity and above a certain length threshold. The algorithm has two variations depending on how we choose the block size. We show that if the block size is kept fixed, the compression performance of the algorithm is similar to that of the greedy algorithm, without the associated expensive space and time requirements. If the block size is varied linearly with the reference file size, the algorithm can run in linear time and constant space. We also show empirically that the algorithm performs better than other state-of-the-art differential compression algorithms in terms of compression and is comparable in speed.*

Introduction

Differential compression, or delta compression, is a way of compressing data with a great number of similarities. Differential compression produces a delta encoding, a way of representing a version file in terms of an original file plus new information. Thus, differential compression algorithms try to efficiently find data common to a reference and a version file to reduce the amount of new information that must be used. By storing the reference file and this delta encoding, the version file can be reconstructed when needed. An overview of differential compression is presented in **Figure 1**.

Early differential compression algorithms ran in quadratic time or made assumptions about the structure and alignment of the data to improve running time. However, many of the applications that we discuss require differential compression algorithms that are scalable to large inputs and that make no assumptions about the structure or alignment of the data.

The advantages of differential compression are clear in terms of disk space and network transmission. The main uses of differential compression algorithms are in software distribution systems, web transportation infrastructure, and version control systems, including backups. In systems in which there are multiple backup files, considerable storage space can be saved by representing each backup as a set of changes to the original (a "delta" file or, simply, a "delta"). Alternately, we can choose to represent the reference file in terms of a version file or represent each file as a set of changes to the subsequent version. Furthermore, transmission costs would be reduced when information was being transferred, since we would need to send only the changes rather than the entire version file. In client/server backup and restore systems, network traffic is reduced between clients and the server by exchanging delta encodings rather than exchanging whole files [1]. Furthermore, the amount of storage required in the backup server is less than if we were to store the entire file. Software updates can be performed this way as well. Also, this method of updating software has the added benefit of providing a level of piracy protection, since the reference version is required in order to reconstruct the version file. Thus, software updates can be performed over the web, since the deltas are of very little use to those who do not have the original.

0018-8646/06/$5.00 © 2006 IBM

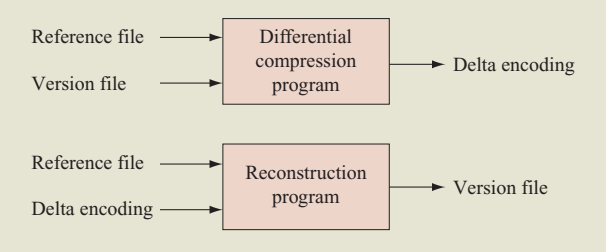

Figure 1

Overview of differential compression.

The problem of differencing can be considered in terms of a single version file along with a reference file or a number of version files along with a single reference file. The *hsadelta* encoding algorithm allows us to bring into memory only the metadata information (offsets and lengths of the copy tokens from the reference file, and lengths of the insert tokens for the new information) about the delta file. This metadata information is usually much smaller and therefore fits in memory. This allows us to reconstruct only a desired part of the delta file. The feature can be used in version control systems for keeping previous deltas and reconstructing any old revision by simply bringing in the metadata and reference file. For purposes of simplicity, we consider the case of a single version file along with a single reference file, but our results can be extended to the consideration of a number of version files along with a single reference file.

Related work

Differential compression arose as part of the string-to-string correction problem [2], finding the minimum cost of representing one string in terms of another. The problem goes hand in hand with the longest common subsequence (LCS) problem. Miller and Myers [3] presented an algorithm based on dynamic programming, and Reichenberger [4] presented a greedy algorithm to optimally solve the string-to-string correction problem. Both of these algorithms ran in quadratic time and/or linear space, which proved to be undesirable for very large files.

One of the most widely used differencing tools is the *diff* utility in UNIX**. It is not desirable for most differential compression programs because it does not find many matching substrings between the reference and version files. Since it does line-by-line matching, it operates at a very high granularity, and when data becomes misaligned (because of the addition of a few characters as opposed to a few lines), *diff* fails to find the matching strings following the inserted characters.

In the past few years, there has been much work in the area of the copy/insert class of delta algorithms. The *hsadelta* algorithm (or, simply, *hsadelta*) falls into this class of algorithms. The copy/insert class of delta algorithms use a string-matching technique to find matching substrings between the reference file and version file, encode these as copy commands, and then encode the new information as an insert command. The *vcdiff* [5] and *xdelta* [6] algorithms and the work of Ajtai et al. [7] fall into the copy/insert class of delta algorithms. The benefit of using this method is that if a particular substring in the reference file has been copied to numerous locations in the version file, each copied substring can be encoded as a copy command. Also, if a contiguous part of the reference file substring has been deleted in the version file, the remaining part of the old substring can be represented as two copy commands. There is a benefit to representing the version file in terms of copy commands as opposed to insert commands. Copy commands are compact in that they require only the reference file offset and the length of the match. Insert commands require that the length as well as the entire new string be added. Thus, for a compact encoding, it is desirable to find the longest matches at every offset of the version file in order to reduce the number of insert commands and increase the length of the copy command.

We compare our results empirically to those of the *vcdiff* [5], *xdelta* [6], and *zdelta* [8] algorithms. These seem to be the most competitive differential compression algorithms currently available in terms of both time and compression. The *vcdiff* algorithm runs in linear time, while the *xdelta* algorithm runs in linear time and linear space (although the constant of the linear space is quite small). The *zdelta* algorithm is a modification of *zlib library*, and it uses a hash-based mechanism on reference and version files. Thus, it uses constant time and space at the cost of compression for very large files. Ajtai et al. [7] present a family of differential compression algorithms for which they prove that one of their algorithms runs in constant space that also has linear time and good compression (empirically). However, in some situations, it produces suboptimal compression. They present three more algorithms that cannot be proven to be linear-time but produce better compression results.

The *vcdiff* algorithm [5], developed by Tichy, Korn, and Vo, combines the string-matching technique of the Lempel–Ziv'77 algorithm [9] and the block-move technique of Tichy [10]. The *vcdiff* algorithm uses the reference file as part of the dictionary to compress the version file. Both the Lempel–Ziv algorithm and the Tichy block-move algorithm run in linear time. The application of differential layer encoding in progressive

transmission using the Lempel–Ziv algorithm has been discussed by Subrahmanya and Berger [11]. Wyner and Ziv have demonstrated that a variant of the Lempel–Ziv data compression algorithm for which the database is held fixed and is reused to encode successive strings of incoming input symbols is optimal, provided that the source is stationary [12]. Gibson and Graybill have discussed the application of hashing for the Lempel–Ziv algorithm [13]. The *vcdiff* algorithm allows string matching to be done both within the version file and between the reference file and the version file. In the *vcdiff* algorithm, a hash table is constructed for fast string matching.

In his master's thesis, MacDonald has described the *xdelta* algorithm and the version control system designed to utilize it [6, 14]. The algorithm is a linear-time, linear-space algorithm and was designed to be as fast as possible, despite suboptimal compression [14]. It is an approximation of the quadratic-time linear-space greedy algorithm [1]. The *xdelta* algorithm works by hashing blocks of the reference file and keeping the entire hash table in memory. When there is a hash collision, the existing hash value and location are kept, and the current hash value and location are discarded. The justification for this is to favor earlier, potentially longer matches. Both hash values and locations cannot be kept if the algorithm is to run in linear time, since searching for matches among the duplicate hash values would cause the algorithm to deviate from linear time. At the end of processing the reference file, the *fingerprint table* is considered populated. The version file is processed by computing the first hash value on a fixed-size block. If there is a match in the hash table, the validity of the match is checked by exact string matching, and the match is extended as far as possible. Then the version file pointer is updated to the location immediately following the match. If there is no match, the pointer is incremented by one. The process repeats until the end of the version file is reached. The linear space of *xdelta* appears not to be detrimental in practice, since the constant is quite small [14]. It does not find the best possible matches between the reference file and the version file, since hash collisions result in new information being lost. Each subsequent hash to the same location is lost, and the previous information remains in the hash table.

Ajtai et al. [7] have presented four differential compression algorithms: the one-pass differencing algorithm, the correcting one-pass algorithm, the correcting 1.5-pass algorithm, and the correcting 1.5-pass algorithm with checkpointing. Their one-pass differencing algorithm has been proven to be linear in the worst case but to produce suboptimal compression, since it neither detects transpositions in data nor finds optimal matches at a given location of the version file. Their one-pass algorithm continually processes both the original file and the version file sequentially, finding matches by hashing blocks and comparing the blocks. Thus, their algorithm encodes the first match it sees and then clears the hash tables. Hence, all encoded matches must be in the same sequential order between the reference file and the version file to be detected. In order to address these shortcomings, they have devised two new methods, corrections and checkpointing. Their corrections method is a way to improve the match, yet it does not guarantee that they pick the optimal match. It involves implementing a circular queue to store previous hash values of the reference file; thus, it can also cause the existing one-pass algorithm to deviate from the worst-case linear-time complexity. The 1.5-pass algorithm works by first hashing footprints in the reference file, but when there is a hash collision it stores only the first offset encountered. Next, the version file is continually processed by hashing on blocks from the version file. If there is a valid match, the match is encoded, the version file pointer is incremented, and the process continues. The checkpointing method is used when all possible hash values cannot fit into memory and a subset of these hash values must be selected as an accurate representation of the file. Thus, checkpointing implemented along with corrections allows the existing algorithm to improve upon matching substrings already found. This modified algorithm can find longer matches. The work of Ajtai et al. is currently used in the IBM Tivoli* Storage Manager product.

Differential compression methods have also been presented that are based solely on suffix trees. Weiner [15] has proposed a greedy algorithm based on suffix trees that solves the delta encoding problem using linear time and linear space. In contrast to *xdelta*, the constant factor of this algorithm is quite large, preventing it from being used on very large files. Our work combines the hashing techniques of Ajtai et al. [7] and those of MacDonald [6] with the suffix array methods of Manber and Myers [16].

In the most recent work in differential file compression, Shapira and Storer present a differential compression algorithm that works in place and uses the sliding-window method [17]. They show through empirical results that the method is effective in tackling the differential compression problem. Their algorithm uses $O[\max(n, m)] + O(1)$ space, where m is the size of the reference file and n is the size of the version file. The $O(1)$ factor comes from the amount of space needed to store the program and a fixed number of loop variables, etc. They also show through empirical results that the limited amount of memory does not impair the compression performance of their algorithm.

Our contributions

The *hsadelta* algorithm, presented in this paper, can be regarded as an approximation to the greedy algorithm for

differential compression or as a linear-time constant-space differential compression algorithm that is scalable to very large files, depending on how we define our granularity, or block size. Previous differential compression algorithms used hashing on blocks of fixed width or a sliding-window method or suffix trees or LZ'77 with block move. Our work combines the use of hashing on blocks of a fixed width with suffix arrays, a data structure similar to suffix trees that requires less space. Unlike previous algorithms that hash on blocks, *hsadelta* finds the best match at every offset of the version file encountered. In this paper we also introduce three new data structures that significantly reduce the cost of matching version file hash values against reference file hash values.

The *hsadelta* algorithm

Preliminaries

In this section we introduce some notation that is used throughout the remainder of the paper. We define some quantities and functions to help us describe the algorithms to be discussed:

- m is the length of the reference file.
- n is the length of the version file.
- p is the width of each block.
- l is the number of blocks in the reference file, $l = (m/p)$.
- $H(x)$ is the value of the hash function for a block of data that starts at location x and ends at location $x + p - 1$.
- $lcp(x, y)$ is the longest common prefix of two substrings x and y.
- s is the number of bits to index the pointer array; note that $s = \lceil \log_2 l \rceil - 2$.
- t is the number of bits to index the quick index array; note that $t = \lceil \log_2 l \rceil + 5$.

We also define some pointers necessary for the algorithm pseudo-code:

- v_c is the position of a pointer in the version file.
- v_{temp} is a temporary pointer to the version file.
- r_c is the position of a pointer in the reference file.
- v_{prev} is pointer to the version file used to denote the end of the previous encoded match.

Algorithm overview

The *hsadelta* algorithm, much like previous algorithms, hashes on blocks of the reference and version file. The hash value is an abbreviated representation of a block, or substring. It is possible for two different blocks to hash to the same value, giving a spurious match. Therefore, if the hash values match, we must still check the blocks using exact string matching to ensure that we have a valid match. It also follows that hash values do not uniquely define a block, but if two blocks have different hash values, the two blocks do not match. We use the Rabin–Karp hash method [18] because we believe it produces the smallest number of collisions in most situations and it is efficient for computing for a sliding window. Furthermore, we hash *modulo* a Mersenne prime, $2^{61} - 1$. The reason we use a Mersenne prime is that it allows us to compute hash values very efficiently. All hash values are 61-bit numbers stored in a 64-bit (8-byte) unsigned long integer. Typically the number of blocks in the reference file is much smaller than $2^{61} - 1$. Thus, only a small subset of all possible hash values are present in the reference file. This reduces the probability of a spurious match when two unequal blocks hash to the same value.

The algorithm also takes advantage of the suffix array data structure introduced by Manber and Myers [16]. The suffix array is a data structure that can be used for online string matching, yet requires three to five times less space than a suffix tree. The suffix array construction is detailed in Appendix A and is similar to that of Manber and Myers [16].

We must also choose a block size (or seed length [7]). The block size is a very important parameter because it determines the granularity of the matches we detect as well as the length of the shortest match detected by the algorithm. It also governs the amount of memory needed for the algorithm, which is inversely proportional to the block size.

In *hsadelta*, we first process the reference file. Once this is done, we have a complete hash table that is stored in memory. We store this information as the *hash value array*, which is an array of l eight-byte values. After we have completed these hash computations, we create a suffix array of 29 high-order bits of hash values. The size of the suffix array is $2 \cdot l$, where l is the number of hash values. The exact creation of the suffix array is described in Appendix A. We also create three data structures that allow for easy search into the hash value array. These data structures are described in more detail in the next section. We next process the version file by starting at offset zero and computing the hash value at every offset until we find a match. If we have a match, *hsadelta* is structured to pick the longest match with an error of approximately a block length. In order to ensure that we have the best match, we keep track of the best seen match of the version file until we are done hashing on every offset of that block. When we encounter a better match, we replace the match. This procedure ensures that we have examined all possible relative alignments between the two files. We increment the pointer to the location immediately following the match and continue the same

process until we are done processing the entire reference file. The run time of *hsadelta* is improved by the creation of three data structures, as discussed in the next section.

Important data structures

In this section, we introduce three data structures that are crucial to the running time of *hsadelta* but do not affect its complexity analysis. We have found that the construction time of these data structures is fairly small in the entire algorithm, but the data structures improve the running time. The three data structures are the *quick index array*, the *block hash table*, and the *pointer array*. These three data structures are created in one pass of the suffix array constructed by methods described in Appendix A. For the purpose of discussing *hsadelta*, it is important to remember that the suffix array allows for quick searching. The array is essentially a set of indices in the hash value array such that the hash substring starting at an index given early in the suffix array is lexicographically lower than a hash substring starting at an index given later in the array. This is consistent with the definition provided by Manber and Myers [16].

The purpose of the quick index array is to serve as an early exit mechanism if a hash value is not present in the hash table. Use of the quick index array is important because there are 2^{61} possible hash values given the prime number used in our hash computation. The number of distinct hash values is always less than the number of blocks, or l, which is typically less than 2^{24}. This means that our hash table is about $7 \times 10^{-12}\%$ populated. Thus, it is to our advantage to design data structures that take advantage of the sparsity of the hash table. The size of the quick index array can be described as a tradeoff between storage and the efficiency of our early exit scheme. It is an array of bits that contains 2^t bits or 2^{t-3} bytes (see the section on preliminaries for variable definitions) and is initialized to contain all 0s.

The quick index array is formed by a single pass on the suffix array. For each hash value in the suffix array, we extract the first t bits ($t = \lceil \log_2 l \rceil + 5$) of the hash value. We use those bits to index the location in the quick index array and place a 1 in that bit location. Thus, if a hash value is present in the suffix array (and thus in the hash value array), there will be a 1 present in the location indexed by the first t bits of the hash value. If a hash value is not present in the suffix array, there could be either a 1 or a 0 in the location indexed by the first t bits of the hash value, depending on whether there is another hash value present with the same leading t bits. Thus, the quick index array serves as an early exit mechanism if the hash value we are searching for is not in the hash table.

We decided on the number of bits to use by the following reasoning: Since there are at most l different hash values, at most l of the 2^t bits in the hash table are

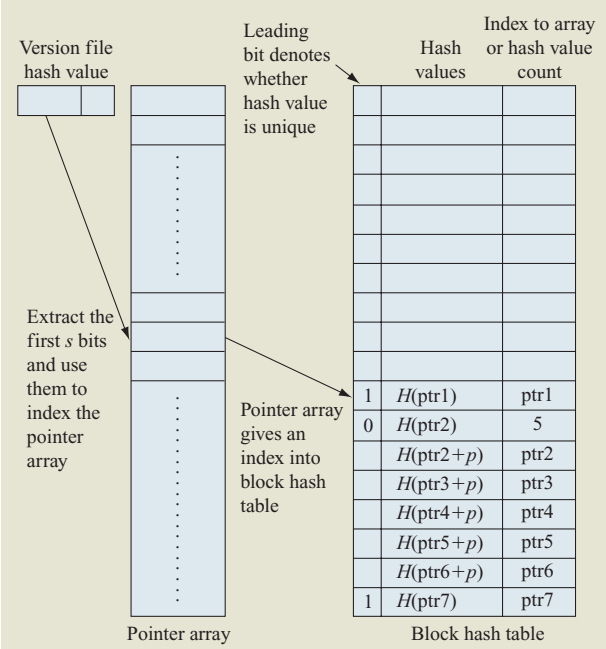

Figure 2

Block hash table illustrating how we access a value in the pointer array using the first s bits of the version file hash value. This value in turn produces an index to the table. A few typical entries are shown.

1s. Thus, for a random hash value that is not present in the hash value array, the probability of finding a 1 in the quick index array is less than 2^{-5}. As can be seen, that quick index array is a special case of a *Bloom filter* [19], for which k (the number of hash functions) is 1 and m/n (where m is the table size and n is the number of keys) is 32 in *hsadelta*. This gives us a 97% rejection rate of version file hash values. Increasing k to 2 or more can reduce the filter size at the cost of a proportional increase in the number of cache misses. For a quick index array that does not fit in the cache, the cost of checking a hash value in the array is k cache misses. The cost of one cache/TLB miss is approximately 500 cycles on our machine. This affects the performance of the algorithm with an improvement of only 2% in the rejection ratio. Since we make only one pass on the suffix array, our data structure takes $O(l)$ time to construct, where l is the number of hash values in the hash array.

Our second data structure is the block hash table, as illustrated in **Figure 2**. In the block hash table, we organize our reference file hash values for efficient matching. The block hash table is implemented as an array. The even locations in the array contain the (29-bit) high-order hash values in the same order as they appear in the suffix array, and in the odd locations we have either

an index to hash value array or a hash value count. We also want to distinguish between unique hash values and hash values that are replicated, since we need to know whether we should perform binary search to find the longest match. Since our hash values are 61 bits and we use 29 high-order bits, we have three free bits available. We use the highest bit to distinguish between unique and duplicate hash values. The leading bit is a 1 if the hash value is unique and a 0 if the hash value is not unique. The remaining bits contain the high-order 29 bits of the hash value $H(r)$, where r is a pointer in the reference file. If $H(r)$ is unique, the odd location contains its index r in the hash value array. If $H(r)$ is not unique, the first odd location contains the count, or total number of occurrences of this hash value in the suffix array. The first pair after the hash value and count represents the lowest-ranked substring that starts with $H(r)$, and the last pair represents the highest-ranked substring that starts with $H(r)$. In the example of Figure 2, ptr1 to ptr7 entries are indices of the hash value array. In the figure, the hash value string at ptr2 is the lexicographically lowest string starting with H(ptr2), and the hash value string at ptr6 is the lexicographically highest string starting with H(ptr2).

All of the substrings are ordered as they are in the suffix array, since the block hash table was created by a single pass of the suffix array. Thus, every hash substring starting with $H(r)$ is in its lexicographic order. Each pair consists of a pointer to the hash value array in the odd locations, and the corresponding even positions contain the high-order 29 bits of the hash value corresponding to the next block after the pointer. This provides immediate access to high-order hash values for comparisons in a binary search. However, we need to access the hash value array to find the exact matches and perform a search in case high-order values match. Unique and duplicate hash values are represented as indicated in the figure.

The final data structure is the pointer array. The pointer array is an array of indices to the block hash table that is indexed by the first s, or $\lceil \log_2 l \rceil - 2$, bits of the hash value. This location contains a pointer to the smallest reference file hash value with the same leading s bits. If there is no reference file hash value with the same leading s bits, the location contains the next lowest reference file hash value. Thus, the pointer array reduces the number of hash values we must process in order to determine whether a match exists.

We decided on the number of bits to use by the following reasoning: There are at most l different hash values in the hash table; if we use s bits, each pointer in the pointer array, on average, will map to approximately four distinct hash values in the block hash table, assuming that we have a good hash function. The pointer array and block hash table combination serves as a two-dimensional table of hash values. The new hash function is the choice of leading s bits. Given a good first-61-bit hash function, we expect that the hash value bits are also random with good uniform distribution. We should expect no more collisions through choosing first s bits than by any other s-bit hash computation on the hash values.

These three data structures are used for efficient matching between the version file hash values and the reference file hash values. This matching algorithm is shown in **Figure 3**. The matching is done as follows: After each hash value computation in the version file, we extract the first t bits and check the quick index array. If it is a 0, we quit because there is no match in the hash value array. If it is a 1, we check the pointer array. The first s bits of the version file hash value give us an index in the block hash table. If the high part (high-order 29 bits) of the hash value to which it points in the block hash table is greater than the high part of our version file hash value, we know that the version file hash value is not in the block hash table. We exit, since we do not have a match. If the high part matches, we obtain the full hash value from the block hash array index pointed to from the block hash table. If this hash value is greater than the version file hash value, we quit. If the hash value is equal to the version file hash value, we check to see whether the match is unique. If the match is unique, we return the match. If the match is not unique, we perform a binary search to see which of the matches provides the longest match.

The nature of the suffix array allows us to perform binary search, since the hash substrings with the greatest number of block matches are grouped together. After we have found the longest match, we return it. If the high part of the hash value to which the pointer array points is less than our current high part of the hash value, we sequentially traverse other hash values in the array to see whether we have a match. If the hash values that we process sequentially are repeated hash values, we use the count in the odd entries to skip over the next count pair of values. As mentioned earlier, we have approximately four distinct hash values mapped to one pointer; thus, the sequential processing of hash values is not a time-consuming step. Also, if we have repeated hash values, this does not affect the complexity of the sequential processing of hash values, since the count value allows us to skip over the repeated hash values. These data structures are illustrated in Figure 2. If we encounter a hash value that is larger than our desired hash value while processing the block hash table sequentially, we quit. If we encounter a match on the high part, we follow the same matching procedure as described above for the complete hash value. As one can see, the quick index array, the pointer array, and block hash table substantially reduce the amount of time spent searching for the desired hash substring. In the absence of these data structures, for all version file hash values generated,

Input: hash value $H(v_c)$ that we are seeking to match:

1. if [the corresponding entry to $H(v_c)$ in the quick index array is 0]
   (a) return a null value since $H(v_c)$ is not in the hash table
2. else
   (a) use the corresponding entry in the pointer array to get a hash value in the block hash table
   (b) if [high part (hash value) in the block hash table > high part $H(v_c)$]
      i. return a null value since $H(v_c)$ is not in the table
   (c) else if high parts are equal
      i. get the hash value from the hash value array pointed to by the block hash table
      ii. if [the hash value > $H(v_c)$]
         A. return a null value since $H(v_c)$ is not in the table
      iii. else
         A. now sequentially process the values in the block hash table (high part and low part from the hash value array) until we either find an entry equal to $H(v_c)$ or we find the first entry that is larger than $H(v_c)$
         B. if the latter case is true
            • return a null value because $H(v_c)$ is not in the table
         C. else (we have a match)
            • if the match is unique, indicated by the leading bit of the 32-bit entry in the block hash table
              - return the match
            • else (the match is not unique)
              - perform binary-search-for-suffixes as described in Figure 5 to find the longest match where the initial range is count number of values
              - return the longest match
   (d) else
      i. do as in (c)iii [in this case high part of the hash value in the block hash table < high part of $H(v_c)$]

Output: the best match for the input hash value, $H(v_c)$, if it exists and a null value otherwise.

Figure 3

Best-match algorithm—Finds the best match for a particular version file hash value in the hash table.

Input: reference file string, version file string:

1. (a) initialize $r_c = 0$.
   (b) while ($r_c < m - p$)
      i. compute $H(r_c)$ and store in the *hash value array*
      ii. increment r_c by p
2. Call suffix-array-construction with *hash value array* as input.
3. In a single pass of the suffix array, create the quick index array, the pointer array, and the block hash table as described in the section on important data structures.
4. Initialize $v_c = 0$ and $v_{prev} = 0$.
5. While ($v_c < n - p$)
   (a) initialize best-seen-match to null
   (b) compute $H(v_c)$
   (c) call best-match-algorithm as described in Figure 3 with $H(v_c)$ as input
   (d) if matching algorithm returns null
      i. increment pointer v_c by 1
   (e) else (we have a match)
      i. store match in the best-seen-match variable
      ii. $v_{temp} = v_c$
      iii. for ($v_c = v_{temp} + 1$ to $v_c = v_{temp} + p - 1$)
         • compute $H(v_c)$ and call the best-match-algorithm with $H(v_c)$ as input
         • if matching algorithm returns a match and it is longer than the best-seen-match
           - update the best-seen-match variable to contain the current match
      iv. check the validity of the best-seen-match and extend it as far left and right as possible and finally encode the match as a copy command
      v. encode the information starting at v_{prev} to the start of the match as an insert command
      vi. update the v_c and v_{prev} pointers to the end of the match
6. Encode the last bit of information from v_{prev} to the end of the file as an insert command.

Postcondition: delta encoding with copy and insert commands and new information to be encoded.

Figure 4

Main differential compression algorithm.

we have to perform binary search on the entire suffix array, which contains l values.

Algorithm pseudo-code

The main differential compression algorithm is shown in **Figure 4**. The specific binary search method (hereafter designated simply as *binary search*) that we use on the suffixes in the block hash table is shown in **Figure 5**. This binary search method is similar to that presented in the work of Manber and Myers [16]; it is a modification of the familiar binary search method. We maintain a *lowRankPtr* and a *highRankPtr*, which are pointers that index the block hash table. In the example given in Figure 2, the lowRankPtr is initially set to the row containing ptr2 and the highRankPtr is set to the row containing ptr6.

Note that we can reduce the number of hash value comparisons in our binary search method by keeping track of min($|lcp(low, w)|$, $|lcp(w, high)|$), where *low* is the string in the hash value array corresponding to lowRankPtr and *high* is the string in the hash value array corresponding to highRankPtr. This binary search method exploits a property of suffix arrays in which, when searching for a particular string w in a suffix array, if the length of the longest common prefix is x blocks in

Precondition: We have a range of suffixes that match our version file hash value string in the first block. Let the first block of the version file hash value be $H(v_c)$.

1. Initialize our *lowRankPtr* to point to the lexicographically lowest reference file hash substring starting with $H(v_c)$ (the first pair of values after $H(v_c)$ and its *count*; this is the row containing ptr2 in the example given in Figure 2) and *highRankPtr* to point to the lexicographically highest reference file hash substring starting with $H(v_c)$ (count pair of values after $H(v_c)$ and its count; this is the row containing ptr6 in the example given in Figure 2). Let w be the string of hash values that we seek to match.
2. While (*highRankPtr* − *lowRankPtr* > 1)
   (a) *midRankPtr* = (*highRankPtr* + *lowRankPtr*)/2
   (b) *Ptr* ← the pointer stored in the odd location of the *midRankPtr* row of the block hash table
   (c) if the string of hash values starting at hash value array[*Ptr*] > w then
      i. *highRankPtr* = *midRankPtr*
   (d) else
      i. *lowRankPtr* = *midRankPtr*
3. Compare |*lcp*(*low*, w)| and |*lcp*(w, *high*)| where *low* is the string in the hash value array corresponding to *lowRankPtr* and *high* is the string in the hash value array corresponding to *highRankPtr* to find whether *low* or *high* is the longer match.
4. Return the longer of the matches from the previous step

Postcondition: We have a single suffix that matches the version file hash value string in the maximum number of blocks.

Figure 5

Binary search among sorted suffixes.

length, all suffixes that match w to x blocks will be adjacent to one another in the array. For the in-memory implementation of the algorithm, we have the optimization that rather than extending the best seen match after examining all the offsets of a given block, as described in Step 5(e)iv of Figure 4, we check the validity of the match for each offset and then extend the match on either side in Step 5(e)iii. Since our data is all in memory, this step is not expensive. The step may give us a longer match, up to two blocks in length. Also, after the binary search, the best match for that offset is compared against the best seen match, on the basis of the total length of the match.

Time and space analysis

In this section, we analyze the time and space requirements of the *hsadelta* algorithm. It then becomes obvious that depending on how we pick our block size, we have either a linear-time constant-space algorithm or an approximation to the greedy algorithm that requires less space and time.

Space analysis

The main data structures in the algorithm that require space proportional to the number of blocks in the reference file are the *hash value array*, the *suffix array*, the *rank array* in the suffix array construction routine, the *quick index array*, the *pointer array*, and the *block hash table*. It is important to mention that the block hash table is created in place using the space occupied by the suffix array and the rank array. The hash value array contains l hash values, each an eight-byte value. The suffix array contains l indices in the odd locations and l hash values (high part only) in the even locations, each a four-byte value. As mentioned in the section on important data structures, the quick index array contains $2^{\lceil \log_2 l \rceil + 5}$ bits or $2^{\lceil \log_2 l \rceil + 2}$ bytes, and the pointer array contains $2^{\lceil \log_2 l \rceil - 2}$ entries or $2^{\lceil \log_2 l \rceil}$ bytes. The block hash table contains $2 \cdot l$ entries if all of the hash values are unique. When the hash values are not unique, the block hash table contains at most $3 \cdot l$ values. This happens when all hash values occur exactly twice. In this situation, we have one row indicating the count, followed by the two rows corresponding to the hash values. The rank array in the suffix array construction routine contains l indices, each of which is a four-byte value. All other variables and arrays require $O(1)$ space. The *hsadelta* algorithm is structured such that the block hash table shares memory with the suffix array and the rank array. Thus, the worst-case memory requirement is only $30 \cdot l$ bytes. The space requirements of the algorithm are summarized in **Table 1**. As can be seen, the space requirements are always less than or equal to $30 \cdot l$ bytes. Note that the requirements for the block hash table are not included in the total, since it uses the space of existing data structures.

The space requirements for the greedy algorithm are simply those of the hash table. However, since the greedy algorithm hashes on every offset of the reference file, there are $n - p + 1$ hash values in the hash table. The space requirements for the greedy algorithm are approximately $3 \cdot n$, since the greedy algorithm as implemented by Ajtai et al. [7] uses a three-byte hash value. Hence, if we choose our block size to be greater than ten bytes (we typically do so in practice), *hsadelta* requires less space than the greedy algorithm. Note that if we increase block size, or p, as the reference file size increases, *hsadelta* requires constant space, because l is constant. In this situation, however, the greedy algorithm is still a linear space algorithm.

Time analysis

We consider a simplified version of *hsadelta* when performing a time analysis. Rather than using the quick index array, the pointer array, and the block hash table in our time analysis, we simply consider matching to be done as a binary search on the entire suffix array. This is reasonable, since the quick index array, pointer array, and block hash table are simply implemented to make the matching process more efficient; hence, we know that the

algorithm performs at least as well as its simplified version. However, we have included the creation of these data structures in its complexity analysis. Another assumption we make is that there are no spurious multi-block matches, that is, instances in which two or more consecutive hash values match but the blocks themselves do not match. This is reasonable, since the probability of having a spurious multi-block match is very low.

During the time analysis, we refer to Figure 4. Although the analysis pertains to a simplified model of our differential compression algorithm, the steps in Figure 4 remain the same. However, the matching algorithm described in Figure 3 is simplified to a binary search on the entire suffix array. The single pass of the reference file and the hash computation in Step 1 of Figure 4 require $O(m)$ time, since the hash computation requires the use of every byte in the reference file. The creation of the suffix array requires $O(l \log l)$ time, as described in Appendix A. The single pass of the suffix array to create the quick index array, the pointer array, and the block hash table requires $O(l)$ time. Now let us consider Step 5 of Figure 4, processing of the version file. In Step 5(b), we could compute hash values at every offset of the version file, a process that has complexity $O(n)$. This argument relies on a property of the Karp–Rabin hash computation [18], which requires $O(1)$ computation on $H(x)$ to produce $H(x+1)$. The binary search routine described in Figure 5, used in the processing of the version file, has complexity $O(d + \log l)$, where d is the depth of the longest match in terms of blocks, as shown in the work of Manber and Myers [16]. We consider two separate cases. The first case is Step 5(d) in Figure 4, in which there is no match to the hash value of a version file offset. In this case the depth is 0, so $d = 0$. Thus, for each version file offset for which there is no match, the complexity is simply the cost of performing a binary search, or $O(\log l)$. In the case in which there is a match [Step 5(e) of Figure 4], we check the matches corresponding to the next $p - 1$ offsets; in other words, we perform a binary search for the hash value generated for each version file offset in this block. This is Step 5(e)iii of Figure 4. Since we must perform a binary search, for each offset within a block, the complexity is $O[p \cdot (d_{\max} + \log l)]$, where $d_{\max}$ is the largest value of d for any offset in the entire block. This results in a match of size $p \cdot d_{\max}$ bytes. Amortized per byte of the match, the complexity is $O[(p \cdot d_{\max} + p \log l)/(p \cdot d_{\max})]$ or $O[1 + (1/d_{\max}) \log l]$. Since $d_{\max}$ is at least 1, this is bounded by $O(1 + \log l)$, or $O(\log l)$. Thus, the worst-case complexity of processing the version file amortized to each byte, regardless of whether or not there is a match, is $O(\log l)$; hence, processing the entire version file takes $O(n \log l)$. The time requirements for the *hsadelta* algorithm are summarized in **Table 2**.

Table 1 Space requirements of each data structure in the *hsadelta* algorithm.

Data structure	*Space (bytes)*
Hash value array	$8 \cdot l$
Suffix array	$8 \cdot l$
Rank array	$4 \cdot l$
Quick index array	Between $4 \cdot l$ and $8 \cdot l$
Pointer array	Between l and $2 \cdot l$
Block hash table	Between $8 \cdot l$ and $12 \cdot l$
Total	Between $25 \cdot l$ and $30 \cdot l$

Table 2 Complexity of each step of the *hsadelta* algorithm.

Step	*Complexity*
Hashing on reference file	$O(m)$
Suffix array construction	$O(l \log l)$
Creation of data structures	$O(l)$
Hashing on the version file	$O(n)$
Processing the version file	$O(n \log l)$
Total	$O(m + l \log l + n \log l)$

As one can see, the total complexity of the algorithm is $O(m + l \log l + n + n \log l)$, which can be reduced to simply $O(m + l \log l + n \log l)$. This can also be written as $O[m + (m/p) \log (m/p) + n \log (m/p)]$. If p remains constant as m increases, the complexity can be simplified to $O(m + m \log m + n \log m)$ or just $O[(m + n) \log m]$. When we consider the algorithm as an approximation to the greedy algorithm by fixing the block size, it clearly has a better asymptotic complexity with similar compression performance, as shown in the next section. In the case in which p increases proportionally with m, (m/p) or l is a constant. Thus, the complexity reduces to $O(m + n)$, and if we increase p proportionally with m (that is, fix l), the algorithm becomes a linear-time constant-space algorithm.

In practice, however, we obtain much better running time than the asymptotic complexity derived above because of the creation of the quick index array, the pointer array and the block hash table, which narrow the scope of our binary search and the matching procedure.

Performance compared to that of the greedy algorithm

The compression performance of the *hsadelta* algorithm is equivalent to the performance of the greedy algorithm for the following reasons. In the greedy algorithm, hash values are computed on the reference string for all offsets. Thus, it can find matching substrings that are at least one block long. The *hsadelta* algorithm computes hash values only on the block boundaries of the reference file. Thus, it

will miss matches that are longer than a block but do not contain a full reference block, or, in other words, if the common substring straddles two blocks. This is because there is no single reference file hash value that represents the information in this match. However, we are sure to find any match that is at least two blocks long, because, in this case, the match contains at least one complete block of the reference file, for which we have computed the hash value. Thus, we can obtain results equivalent to those for the greedy algorithm if we choose our block size to be half of the block size used by the greedy algorithm. As described by Ajtai et al., the greedy algorithm requires storing $n - p + 1$ hash values (approximately n hash values), where n is the length of the reference string. The *hsadelta* algorithm requires the use of only $(2 \cdot n/p)$ hash values, where p is the block size used by the greedy algorithm. In this step, our algorithm reduces the space requirement by approximately $p/2$ without giving up any capability for finding small matches.

When there are multiple matches for a given offset of the version file, the greedy algorithm performs an exhaustive search to find the best match. This is very expensive if hash collisions are frequent. In our algorithm, we perform a binary search, as described in Figure 5, to find the best match with the granularity of a block size. Here our algorithm cannot discriminate between two matches which have same number of full blocks but fractional blocks of different sizes on either side of the match. Other than this flaw, the algorithm is essentially a more efficient implementation of the greedy algorithm. We can circumvent this flaw by modifying the algorithm as follows: After a binary search, we conclude that the longest matching substrings match to a depth of d blocks. We should consider all matches that match at a depth of d blocks as well as those that match at a depth of $d - 1$ blocks. For all of these hash substrings, we look at one block at both ends of the string to find the longest overall match. This is still much more efficient than the greedy algorithm, since we have narrowed down the number of matches for which we perform full string matching. For example, consider the scenario in which there are 100 matches having the same hash values and only four matches that match at a depth of 9 or 10 blocks. The greedy algorithm examines all 100 matches, and each of these examinations may be potentially up to 10 blocks long. In our algorithm, we look at only the four longest matches, and for each of these matches, we have to examine only one block on either side. Note that we have to examine the hash substrings that match at a depth of $d - 1$, since the match may be extended to an additional fractional block on either side to give a match that is longer than d blocks. In other words, with the scenario given above, it is possible that the matches at a depth of 10 match in a length of exactly 10 blocks, and a match at a depth of 9 matches at a depth of almost 11 complete blocks, since the strings could match up to an additional block on either side of the match. However, we do not know this until we actually examine the matches. Thus, it is necessary to explore not only the matches that match in the longest depth, but also those that match in the longest depth minus 1. Our implementation of the differential compression algorithm does not include this optimization.

Experimental results

We compare *hsadelta* with *xdelta*, *vcdiff*, and *zdelta* on a data set ranging in size from 1.3 MB to 1.3 GB. We were not able to compare with the results of the algorithm by Ajtai et al. [7] because the code provided to us did not scale to such large data sets. The data-set corpus is composed of real-world data sets and artificially created data sets in order to demonstrate various aspects of the algorithm. The data-set corpus and the experiments are based on the recommendations of Hunt et al. [5].

We divide the version file into fixed-size buffers (no greater than 20 Mb) and process the buffers as separate files. This method also helps us to optimize for I/O, since we can read in the next version buffer while compressing the first one. The *hsadelta* algorithm uses a runtime test to check the need to compress the data with the second-level *bzip2* compression. If the delta information stream is more than 1 MB in size, we compress the first 1 MB and check whether we have a reduction greater than 5%. If we do, we run *bzip2* on the rest of the buffer. This is done for the delta stream of each version buffer. The process is used to save time for the cases in which a buffer contains encrypted data or already compressed data on which lossless self-compression techniques such as *bzip2* will not provide further compression. If the delta has to be computed on two .tar files,[1] the files are considered as one large file and used as such for the input to the various delta algorithms.

All of the experiments were run on an IBM Intellistation* Model M Pro computer running the Red Hat Linux** v9.0 operating system, with an Intel Pentium** 4 3-GHz processor and 2 GB of RAM. All results were obtained after ensuring that the cache was warm. The timing results were averaged over ten simulations of each compression algorithm after the cache had been warmed up. We ensured that the compression program was the only user-executed program on the CPU, trying to homogenize the environment for running the simulations.

The real-world data set of **Table 3** comprises large software distributions such as linux kernels, gcc, and

[1] A .tar file is a tape archive file; it keeps related files together, thus facilitating the transfer of multiple files between computers. The files in a .tar archive must be extracted before they can be used.

Table 3 Comparison of compression performance of *hsadelta* vs. several other differential compression algorithms for a real-world data set of files. For each entry, the top row gives the size of the delta file in bytes; the lower row gives the compression ratio (version file size/delta size). Missing entries indicate that the algorithm failed to complete or aborted prematurely.

File set	*File size*	*hsadelta*	*xdelta*	*zdelta*	*vcdiff*
netscape-4.78	13,795,752	1,855,840	2,138,632	1,544,408	1,576,305
netscape-4.79	13,800,136	7.44	6.45	8.94	8.75
samba-3.0.4.tar	35,573,760	3,206,433	4,121,002	5,253,316	13,060,041
samba-2.2.9.tar	22,794,240	7.11	5.53	4.34	1.75
gcc-3.3.4.tar	152,770,560	9,323,081	14,944,614	35,483,375	53,727,573
gcc-3.4.0.tar	191,467,520	20.54	12.81	5.40	3.56
gcc-3.4.0.tar	191,467,520	1,376,608	2,896,636	31,633,029	6,244,998
gcc-3.4.1.tar	190,003,200	138.02	65.59	6.01	30.42
linux-2.4.26.tar	172,001,280	14,352,755	20,745,449	43,489,818	89,258,640
linux-2.6.7.tar	197,888,000	13.79	9.54	4.55	2.22
linux-2.6.7.tar	197,888,000	1,859,561	3,256,524	43,530,507	17,387,095
linux-2.6.8.tar	200,847,360	108.01	61.68	4.61	11.55
linux-2.6.8.tar	200,847,360	1,849,226	3,180,391	44,522,144	19,835,278
linux-2.6.9.tar	204,615,680	110.65	64.34	4.60	10.32
gentoo-stage2-x86-2004.1.tar	145,111,040	12,921,779	14,640,011	48,475,530	46,499,676
gentoo-stage2-x86-2004.2.tar	147,281,920	11.40	10.06	3.04	3.17
gentoo-stage3-x86-2004.1.tar	286,556,160	22,787,474	26,264,924	105,214,160	111,180,033
gentoo-stage3-x86-2004.2.tar	293,949,440	12.90	11.19	2.79	2.64
install-x86-universal-2004.0.iso	721,184,768	609,168,451	605,070,346		616,700,030
install-x86-universal-2004.1.iso	706,514,944	1.16	1.17		1.15
potpourri.tar.reference	1,449,656,320	121,207,197	133,135,549		456,428,504
potpourri.tar.version	1,541,416,960	12.72	11.58		3.38
Average compression ratio		40.37	23.63	4.92	7.17

linux OS. The "netscape" distribution is an example of pure object binaries; "gentoo" is a combination of source and object (lisp) files; and "potpourri" is a concatenated tar file of gcc, samba, linux, and gentoo versions. The "potpourri" distribution was created to obtain large files and also to find deltas within distributions. In the table, we compare the compression results. In each set the first file is used as the reference file and the second file is used as the version file. For "samba" we report the result of delta generation from a newer version to an older version; we obtain a 28% higher compression than *xdelta* and an approximately 63% improvement over *zdelta*. Delta compression values between "linux-2.4.26" and "linux-2.6.7" versions are far apart, with significant file size difference. The *hsadelta* algorithm shows 44% better results than *xdelta*. For other gcc and linux results, we see that *hsadelta* outperforms *xdelta* by approximately 80% in compression sizes.

In **Table 4** we report the pristine results of the delta algorithms (i.e., no use of second-level *bzip*- or *gzip*-type compression schemes). As soon as the file size grows beyond few megabytes, the *vcdiff* algorithm quickly degenerates because it fails to find long-distance matches. The *hsadelta* algorithm adaptively selects the block size and the maximum memory used, depending on the available memory in the system or specified by the user and the file size. It never exceeds the 500-Mb memory limit for the experiments. Block size is the smallest match that can be found. The *xdelta* algorithm uses a fixed block size of 16 and the maximum memory available. For medium-sized files (file sizes up to approximately 300 MB), the *hsadelta* results are approximately 30% better than those for *xdelta*. The "potpourri" results demonstrate that if *hsadelta* uses a larger block size to work within the given memory size constraint, a larger pristine output is generated. However, it more than makes up for those losses in the final result by using *bzip2* on the pristine output, as indicated in Table 3.

Table 5 compares the delta and pristine results for *hsadelta* by changing the memory constraint. The pristine output improves by approximately 40% with the increase in memory. The decrease in block size helps *hsadelta* to pick up many small-size matches; however, it does not always translate to better final output because of the generation of a large number of tokens that are not *bzip2*-compressible. The decrease in block size fragments the

Table 4 Comparison of pristine output (output without second level of compression).

File set	*File size*	*hsadelta*	*xdelta*	*vcdiff*
netscape-4.78	13,795,752	3,152,688	4,074,012	1,637,594
netscape-4.79	13,800,136	4.38	3.39	8.43
samba-3.0.4.tar	35,573,760	10,780,947	11,991,996	17,102,081
samba-2.2.9.tar	22,794,240	2.11	1.90	1.33
gcc-3.3.4.tar	152,770,560	34,153,687	44,032,927	112,341,720
gcc-3.4.0.tar	191,467,520	5.61	4.35	1.70
gcc-3.4.0.tar	191,467,520	2,961,812	6,237,463	85,568,046
gcc-3.4.1.tar	190,003,200	64.15	30.46	2.22
linux-2.4.26.tar	172,001,280	43,484,522	53,978,787	119,241,803
linux-2.6.7.tar	197,888,000	4.55	3.67	1.66
linux-2.6.7.tar	197,888,000	4,538,719	7,010,935	118,652,057
linux-2.6.8.tar	200,847,360	44.25	28.64	1.69
gentoo-stage3-x86-2004.1.tar	286,556,160	41,130,998	47,822,564	225,373,687
gentoo-stage3-x86-2004.2.tar	293,949,440	7.15	6.15	1.30
potpourri.tar.reference	1,449,656,320	311,582,836	280,352,480	1,105,258,077
potpourri.tar.version	1,541,416,960	4.95	5.50	1.39

Table 5 Comparison of delta and pristine output for several amounts of memory. For each entry, the upper row shows delta output and the lower row shows pristine output. Results are in bytes.

File set	*File size*	*200 Mb*	*300 Mb*	*500 Mb*
linux-2.4.26.tar	172,001,280	12,921,956	13,686,018	14,352,755
linux-2.6.7.tar	197,888,000	57,234,571	47,756,733	43,484,522
gcc-3.3.4.tar	190,003,200	1,089,494	1,055,198	1,037,190
gcc-3.4.0.tar	191,467,520	3,630,070	3,168,238	2,797,990
gcc-3.4.0.tar	191,467,520	1,413,966	1,381,971	1,376,608
gcc-3.4.1.tar	190,003,200	3,676,793	3,305,645	2,961,812
gentoo-all-stages.1.tar	478,945,280	47,742,049	45,131,635	42,177,936
gentoo-all-stages.2.tar	485,242,880	108,641,860	98,391,567	86,566,329
potpourri.tar.reference	1,449,656,320	128,205,826	123,302,192	121,207,197
potpourri.tar.version	1,541,416,960	383,018,822	355,456,960	311,582,836
Total reference	2,482,073,600	191,373,291	184,557,014	180,151,686
Total version	2,606,018,560	556,202,116	508,079,143	447,393,489

buffer window of the secondary compression algorithms (*zlib*/*bzip2*) which, up to a limit, perform better on bigger windows. Thus, we need a minimum block size above which both delta compression and *bzip2* compression can perform well.

Table 6 shows the performance of *hsadelta* in terms of speed of compression and reconstruction of the data sets. Compression speed is the time taken by the algorithm to compress the version file, so it is version file size divided by time to compress. The "install-x86*.iso" files demonstrate the adaptive capability of *hsadelta* in which it decides against second-level compression (*bzip2*) and shows nearly a factor of 2 improvement in speed over *xdelta* without significantly affecting delta size. The linux and gcc data set provide examples in which *hsadelta* was able to pick up a large number of matches across the file, helping to reduce the delta size and also improve the speed of compression. The last row gives the average compression and reconstruction speeds for various algorithms. We note that *hsadelta* is comparable to *xdelta* in speed. Both of these algorithms are significantly faster than other algorithms, particularly for compressing large files.

Table 7 shows the compression results for jigsaw files. Jigsaw files are created by generating move instructions of random lengths and random source and destination

Table 6 Comparison of compression time and reconstruction time. For each entry, the upper row shows compression time and the lower row shows reconstruction time.

File set	*File size*	*hsadelta*	*xdelta*	*zdelta*	*vcdiff*
netscape-4.78	13,795,752	0 m 3 s	0 m 3 s	0 m 7 s	0 m 10 s
netscape-4.79	13,800,136	0 m 1 s	0 m 0 s	0 m 0 s	0 m 0 s
samba-2.2.9.tar	22,794,240	0 m 19 s	0 m 20 s	0 m 35 s	1 m 10 s
samba-3.0.4.tar	35,573,760	0 m 8 s	0 m 11 s	0 m 11 s	0 m 28 s
linux-2.4.26.tar	172,001,280	2 m 1 s	2 m 22 s	5 m 5 s	12 m 40 s
linux-2.6.7.tar	197,888,000	1 m 3 s	1 m 30 s	1 m 3 s	1 m 28 s
linux-2.6.7.tar	197,888,000	1 m 24 s	2 m 7 s	7 m 30 s	11 m 38 s
linux-2.6.8.tar	200,847,360	0 m 59 s	1 m 28 s	1 m 13 s	1 m 8 s
gcc-3.3.4.tar	152,770,560	2 m 4 s	2 m 17 s	6 m 11 s	12 m 14 s
gcc-3.4.0.tar	191,467,520	1 m 2 s	1 m 27 s	0 m 55 s	0 m 58 s
gcc-3.4.0.tar	191,467,520	1 m 28 s	2 m 15 s	7 m 6 s	7 m 19 s
gcc-3.4.1.tar	190,003,200	1 m 3 s	1 m 37 s	1 m 18 s	1 m 16 s
gentoo-stage2-x86-2004.1.tar	145,111,040	0 m 56 s	0 m 35 s	5 m 53 s	9 m 40 s
gentoo-stage2-x86-2004.2.tar	147,281,920	0 m 16 s	0 m 15 s	0 m 10 s	0 m 25 s
gentoo-stage3-x86-2004.1.tar	286,556,160	1 m 48 s	1 m 19 s	8 m 1 s	24 m 16 s
gentoo-stage3-x86-2004.2.tar	293,949,440	0 m 31 s	0 m 36 s	0 m 29 s	0 m 49 s
install-x86-universal-2004.0.iso	721,184,768	5 m 06 s	9 m 49 s		106 m 40 s
install-x86-universal-2004.1.iso	706,514,944	0 m 50 s	0 m 43 s		0 m 50 s
potpourri.tar.reference	1,449,656,320	11 m 9 s	8 m 42 s		331 m 12 s
potpourri.tar.version	1,541,416,960	3 m 0 s	2 m 52 s		3 m 16 s
Average compression speed		2.52 Mb/s	2.53 Mb/s	0.72 Mb/s	0.42 Mb/s
Average decompression speed		6.89 Mb/s	6.60 Mb/s	6.4 Mb/s	5.71 Mb/s

Table 7 Comparison of delta sizes for jigsaw file examples. Size is shown in bytes.

File set	*File size*	*hsadelta*	*xdelta*	*zdelta*	*vcdiff*
j1	20,971,520	1,349	2,367	12,746,463	13,807,839
j2	41,943,040	2,415	4,028	25,546,779	27,970,833
j3	83,886,080	4,591	8,090	50,908,745	56,059,744
j4	167,772,160	8,991	15,352	102,237,075	112,265,689
j5	335,544,320	18,291	42,121	204,450,572	224,606,798

offsets within the source file. No new data is inserted, no data is deleted, and no segment of the source file is repeated. This means that the source and target file lengths are same. The number of move instructions ranges from 200 to 2,500, depending on file size. The pristine outputs for all but the *zdelta* algorithm are shown (*zdelta* does not provide a mechanism to obtain pristine output). Ideally the target file should be encoded as move (copy/add) instructions only.

Table 8 gives the details of the artificially created longest common subsequence (LCS) data sets. The performance of the delta algorithm is dependent on the size difference between the pairs of files and the file sizes. LCS files are constructed by first performing a set of deletes at random offsets and of randomly chosen lengths. This is followed by insertions of random lengths at random offsets, which are composed of data that cannot be compressed further. We can see in **Table 9** that the *hsadelta* output closely tracks the difference between the pairs of files, which means that we were able to encode most of the LCS as copy tokens. **Table 10** shows the *hsadelta* compression time and compression speed for two unrelated random files.

We have explored the use of *hsadelta* as a self-compression algorithm. It can take advantage of the long-range redundancies in large files, which are missed by *gzip* or *bzip2* because of their limited buffer of 32 KB. The algorithm is a two-pass algorithm. In the first pass we

Table 8 Details of LCS file sets. The designations 10 and 30 indicate the approximate percentage of difference between version file size and LCS file size. The headings "Insert tokens" and "Delete tokens" refer to the number of insert/delete operations performed on the reference file to create the version file.

File ID	*Ref. file size* *Ver. file size*	*Insert tokens* *Delete tokens*	*LCS* *Difference*
lcs-1–10	3,010,560 3,008,223	2,005 2,185	2,709,387 298,836
lcs-1–30	3,010,560 2,988,514	5,905 7,610	2,107,361 881,153
lcs-2–10	145,111,040 144,821,600	1,979 2,109	130,590,673 14,230,927
lcs-2–30	145,111,040 144,967,694	5,962 7,674	101,576,270 43,391,424
lcs-3–10	601,096,192 600,880,556	2,028 2,163	540,980,917 59,899,639
lcs-3–30	601,096,192 600,162,815	5,983 7,511	420,765,717 179,397,098

Table 9 Comparison of delta algorithms on LCS data sets.

File set	*hsadelta*	*xdelta*	*vcdiff*	*zdelta*
lcs-1–10	314,710	376,700	710,881	517,802
lcs-1–30	924,223	999,280	1,393,938	1,102,254
lcs-2–10	14,250,639	14,338,245	104,989,252	33,560,351
lcs-2–30	43,478,357	43,690,205	121,008,877	69,042,863
lcs-3–10	60,649,460	60,929,690	542,596,966	334,196,106
lcs-3–30	179,458,488	179,992,872	554,146,670	387,507,858

Table 10 Random file processing; files were not compressible and were not related to one another.

Reference file size	419,430,400
Version file size	629,145,600
Compressed file size	629,146,032
Compression time	2 m 47 s
Compression speed	3.6 Mb/s

build the suffix array, and in the second pass we find the matches. The matches are restricted to the blocks that have offset values lower than the current offset value. This restricts the algorithm from being used in streaming input/output mode. Finding matching substrings within a file is like computing an auto-correlation on the file, and it provides useful insights into the structure of large files. The leading algorithm in this space is *rzip* [20], which demonstrates the advantages of delta compression on a single very large file.

Concluding remarks

We have presented *hsadelta*, a differential compression algorithm that finds the optimal matches between a reference file and a version file at the granularity of a certain block size. We have shown that the algorithm operates in linear time and constant space when we increase the block size in proportion to the reference file size, and we have shown empirically that this algorithm provides better compression than *xdelta*, *vcdiff*, and *zdelta* in almost all cases investigated. On the whole, the speed of *hsadelta* is comparable to that of *xdelta*, and it is significantly faster than *vcdiff* and *zdelta*. We regard the main contribution of this work to be the use of a suffix array on hash values to find the longest matches—an approach which is computationally relatively efficient and practical, given memory constraints. Previously, hashing was used on raw data to speed up searches, and suffix arrays/trees were used on raw data to find the longest matches. However, to the best of our knowledge, no previous work has combined these two approaches in a novel way. We expect that our method of finding longest matching strings will be used not only in compression applications but also as a general string-matching technique for web searching and in computational biology.

Appendix A: Suffix array construction

In our differential compression algorithm, we use a suffix array construction algorithm similar to that of Manber and Myers [16]. By means of construction, we form a suffix array of the hash values computed on the reference string. The initial array of hash values is called the *hash value array*.

Sorting algorithm

Before we present our suffix array construction algorithm, it is necessary to describe our sorting algorithm, which is used as a part of our suffix array construction algorithm. Our hash values are eight-byte values, and we sort them four bytes at a time. For four-byte hash values (high part), we can sort the hash values in linear time by making five passes over the array of hash values. We call this sorting algorithm four-byte integer-sort.

The algorithm requires the use of three data structures: a *primary array*, a *work array*, and four *count arrays*. We begin with an array that we call the primary array. It contains $2 \cdot l$ total values, where l is the number of hash values computed on the reference file and each value equals four bytes. We store the index of each hash value in the hash value array in the odd locations, and the hash values themselves in the even locations. We require four count arrays, one for each byte position; each array contains 256 values to store the number of occurrences for each possible byte value. We also have a work array, which contains $2 \cdot l$ values much like the primary array,

but is uninitialized at the beginning of four-byte integer-sort. The algorithm is presented in **Figure 6**.

Since four-byte integer-sort requires five passes on the array, its time complexity is five times the size of the array plus a constant amount of time required to allocate the data structures used. It requires space of two times the size of the array plus the size of the count arrays, which is $4 \cdot 256$ integers, or 4,096 bytes. Note that the four-byte integer-sort routine is an order-preserving sort algorithm. This is a necessary feature for the construction of suffix arrays.

Theorem A.1
At the $(n+1)$th pass of four-byte integer-sort, the data is sorted by the lowest-order n bytes.

Proof
We prove this fact by induction.

Base case
Note that the first pass is used to count the occurrence of various byte values. On the second pass of four-byte integer-sort, the values are placed in buckets on the basis of their lowest-order byte. The count array tells us how large each bucket has to be, progressing from the smallest byte to the largest byte. Thus, the values are sorted on their lowest-order byte.

Inductive step
We assume that on the nth pass of four-byte integer-sort, the data is sorted by the lowest-order $n-1$ bytes. We next consider the $(n+1)$th pass of the algorithm and consider two hash values in the array, c_i and c_j. Without loss of generality, let $c_i \leq_n c_j$, where $\leq_n$ respects ordering only to the last n bytes. We first consider the two cases in which c_i and c_j are not equal. Let $c_i[n]$ be the nth-order byte of c_i.

Case 1
$c_i[n] = c_j[n] \rightarrow ci <_{n-1} c_j$. Thus, after the nth pass, c_i appears before c_j in the partially sorted array. In the $(n+1)$th pass of the algorithm, c_i and c_j are to be placed in the same bucket. However, in the $(n+1)$th pass of the algorithm, c_i is encountered before c_j, and it is placed in the bucket before c_j. Thus, c_i appears before c_j after the $(n+1)$th pass is complete.

Case 2
$c_i[n] < c_i[n]$. This means that on the $(n+1)$th pass, c_i is to be placed in a bucket of smaller value than c_j. Thus, c_i appears before c_j after the $(n+1)$th pass is complete.

Case 3
We now consider the case in which $c_i =_n c_j$. The $(n+1)$th pass of the algorithm seeks to place c_i and c_j in the same bucket. After the nth pass of the algorithm, if there are values c_k that are between c_i and c_j in the partially sorted array, $c_i =_{n-1} c_k =_{n-1} c_j$ for all such c_k. Thus, when this array is sequentially processed in the $(n+1)$th pass, c_i is placed in the bucket corresponding to its nth byte, the c_k are placed in the bucket corresponding to their nth byte, and c_j is placed in the bucket corresponding to its nth byte. Hence, if there are values between c_i and c_j after the $(n+1)$th pass of the algorithm, they must equal c_i and c_j in the lowest n-order bytes.

At the end of four-byte integer-sort, the high-order four bytes of the hash value array have been sorted. We scan the values to determine whether there are duplicates. Then, for each set of duplicate values as a group, we again sort on the basis of their lower four-byte values by using the above four-byte integer sort. □

Input: an array (the primary array) with hash values in the even locations and the corresponding pointers to the hash value array in the odd locations.

1. First we construct the four *count arrays*, one for each byte position to be sorted. In the first pass, we count the number of occurrences of each byte value, which will be anywhere from 0 to 255. Thus we have created four arrays with 256 values each that contain the number of occurrences of each byte value. This step takes $O(l)$ time. From now on we refer to the lowest-order byte as byte 1, the next byte 2, etc. until the highest-order byte is byte 4.
2. We sequentially process the *primary array* and copy values into the *work array* using the *count array*. The *count array* for byte 1 tells us exactly where each hash value (along with its index in the hash value array stored in the odd location) should be placed when sorting on the lowest-order byte. After completion of this step, the *work array* is sorted on the lowest-order byte of hash values by Theorem A.1. This step takes $O(l)$ time.
3. Now we sort on the next low-order byte (byte 2), using the array *work array* and the *count array* for byte 2. We sequentially process the *work array* and place the hash values (with their indices in the odd locations) in their correct places in the *primary array*. We use the *count array* which tells us where the hash values should go. At the completion of this step, the *primary array* has the hash values sorted on the last two bytes by Theorem A.1. This step takes $O(l)$ time.
4. Now we sort based on the next low-order byte. We sequentially process the *primary array,* and each hash value is stored in its correct location using the *count array* for byte 3. At the conclusion of this step, the *work array* has the hash values sorted on the last three bytes by Theorem A.1. This step takes $O(l)$ time.
5. Finally we sort based on the highest-order byte, using the same process as described in step 3. At the conclusion of this step, the *primary array* has the hash values sorted on all four bytes by Theorem A.1. This step takes $O(l)$ time.

Output: an array with sorted hash values stored in the even locations and the corresponding pointers to the hash value array in the odd locations.

Figure 6

Four-byte integer-sort algorithm.

Input: array with hash values (high part) in the even locations and corresponding pointers in the odd locations.

1. Sort the hash values as described in Appendix A. At the end of this step, we have sorted the hash values, with the high part stored in even locations of the array. The corresponding pointers in the odd locations have been moved with the hash values as they were sorted; thus, each hash value is still adjacent to its pointer. This is our preliminary suffix array.
2. If all hash values are distinct, the hash values are sorted and we are done. If the hash values are not distinct, we construct and initially populate the *rank array*. The *rank array* is initially populated by a single scan on the preliminary suffix array. If a hash value is unique, it has a unique rank value. However, if the hash value is not unique, all identical hash values share the same rank. We now define a *group* to be the set of identical hash values. Note that each *group* shares the same rank.
3. Initialize the depth of sorting variable to 1.
4. For each group, we perform the following procedure:
   (a) We sort on the *ranks* corresponding to the location that is equal to the current location plus the depth of sorting. These ranks are obtained from the *rank array*. The sorting is done using the four-byte integer-sort described in Figure 6.
   (b) Then we update the *ranks* in our *rank array*.
5. At the conclusion of step 4, the hash values are sorted at twice the depth of sorting before step 4. Therefore, we double the depth of the sorting variable.
6. If there are still unresolved *ranks*, go to step 4. Otherwise, exit.

Output: suffix array that contains the indices corresponding to the sorted suffixes. In other words, *string of hash values starting at suffix* [i] < *string of hash values starting at suffix* [$i+1$] $\forall i$.

Figure 7

Suffix array construction algorithm.

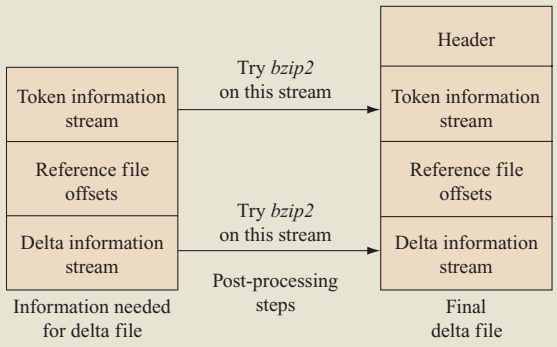

Figure 8

Encoding scheme and post-processing steps.

Suffix array construction algorithm

At the start of the suffix array construction algorithm, we sort the eight-byte hash values as described above. At the end of this process, if the hash values are distinct, we have the suffix array. If the hash values are not distinct, for each group of duplicate hash values, we must look at the subsequent hash values in the hash value array until the strings of hash values are distinct.

We create a data structure called the *rank array* that allows us to access the suffix array position (or the rank) for any location in the hash value array. As we transform the primary array into the suffix array of hash values, the rank array is continually updated. Thus, we index the rank array by the index of the original hash value array, and the rank array value is its location in the suffix array.

Our suffix array construction algorithm is described in **Figure 7**. We know that our suffix array construction terminates, because no two strings are the same in the hash value array. Our suffix array construction takes $O(l \log d)$ time, where l is the number of hash values (or blocks) and d is the length of the longest matching substring (in blocks) in the reference file.

Appendix B: Encoding scheme

We divide the version file into fixed-size buffers that can be brought into memory for compression. The size of these buffers is decided on the basis of the maximum memory available for compression with the limit of 20 Mb. In order to have a compact representation of these version file buffers, we have developed an encoding scheme that we believe is not only compact but is further compressible, since all of the delta substrings are concatenated as a single string.

Our encoding scheme, depicted in **Figure 8**, is described below. We first describe the information we must represent in our delta file. We describe matching substrings between two files as *tokens*. We have two kinds of tokens, *reference file tokens* and *delta file tokens*. The reference file tokens must store the length of the match and the location of the start of the match in the reference file. This is equivalent to the copy command term used in other work. The delta file tokens must store the length of the new substring to be added. This is equivalent to an add command in other work. Thus, we have three different streams of data, the token information stream (which gives us information about whether we have a delta file token or a reference file token along with the length of the token), the reference file offset stream (which tells us where the matching substring starts in the reference file), and the delta stream (the substrings that must be added). The separation of token streams helps us to separate metadata information from the deltas. Thus, by just bringing the metadata information into memory, we are able to reconstruct only a portion of the file, which can be useful for keeping previous deltas in version control systems.

The token information stream is stored as follows: The first bit denotes whether the token is a delta file

token or a reference file token. The next six bits contains the length. If six bits is not enough information to store the length of the match, we terminate the byte with a 1 in the last bit. If six bits is sufficient, we terminate the byte with a 0 in the last bit. If we need to use the next byte, the first seven bits is used for length (which gives a total of 13 bits for length) and the last bit tells us whether we have used the next byte or the token has been terminated. We know that the token terminates when the last bit of a byte is a 0. Thus, the length of the token contains a certain number of complete bytes. The rationale behind this scheme is that most tokens can be represented with only one byte. In rare cases, we need to use two or more bytes to represent the length. Also, we want to represent tokens of small length compactly in order to maximize the reduction in space.

The reference file offset stream is stored using a fixed number of bits per token, namely $\lceil \log_2 m \rceil$, since it is the maximum possible offset value. The total length of the reference file offset stream is $\lceil (k/8) \cdot \lceil \log_2 m \rceil \rceil$, where k is the number of reference file tokens. Finally, the delta information stream contains all of the new information to be added in one contiguous stream.

In order to reduce the amount of information to be stored, we try using *bzip2* on the token information stream and the delta information stream. If the delta information stream is greater than 1 MB, we compress the first 1 MB and ascertain whether we have achieved more than a 5% reduction. If we have, we run *bzip2* on the rest of the data stream. If not, we realize that running *bzip2* does not help, and we do not continue. If the delta information stream is less than 1 MB, we run *bzip2* on the stream. Note that if the compressed stream is larger than the original stream, we use the original stream for the token information stream and the delta information stream, as the case may be. Since we have found that the reference file offset stream is generally not compressible, we do not try to run *bzip2* on it. Thus, for each version file buffer we have a token buffer stream with token header encoding the length of the stream and *bzip2* status. The other two streams also have similar headers. Then there is the main header, which has the *hsadelta* magic number, the number of version file buffers, and the offsets of version file compressed buffers.

Acknowledgments

The authors thank Randal Burns, Laurent Chavet, Fred Douglis, Dana Shapira, and Andrew Tridgell for providing their code and/or data sets. We thank Dharmendra Modha and Moidin Mohiuddin for their tremendous assistance and helpful conversations. Finally, we also thank the reviewers for constructive criticisms and helpful suggestions that have resulted in a better paper.

References

1. R. C. Burns and D. D. E. Long, "Efficient Distributed Backup and Restore with Delta Compression," *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems,* 1997, pp. 27–36.
2. R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *J. ACM* **21**, 168–173 (1974).
3. W. Miller and E. W. Myers, "A File Comparison Program," *Software Pract. & Exper.* **15**, 1025–1040 (1985).
4. C. Reichenberger, "Delta Storage for Arbitrary Non-Text Files," *Proceedings of the 3rd International Workshop on Software Configuration Management,* 1991, pp. 144–152.
5. J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta Algorithms: An Empirical Analysis," *ACM Trans. Software Eng. & Methodology* **7**, No. 2, 192–214 (1998).
6. J. P. MacDonald, "Versioned File Archiving, Compression, and Distribution," University of California at Berkeley; see *http://citeseer.ist.psu.edu/macdonald99versioned.html* (1999).
7. M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly Encoding Unstructured Input with Differential Compression," *J. ACM* **49**, No. 3, 318–367 (2002).
8. D. Trendafilov, N. Memon, and T. Suel, "Zdelta: An Efficient Delta Compression Tool," *Technical Report TR-CIS-2002–02*, Polytechnic University, 2002.
9. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **IT-23**, 337–343 (1977).
10. W. F. Tichy, "The String-to-String Correction Problem with Block Move," *ACM Trans. Computer Syst.* **2**, 309–321 (1984).
11. P. Subrahmanya and T. Berger, "A Sliding Window Lempel–Ziv Algorithm for Differential Layer Encoding in Progressive Transmission," *IEEE Trans. Info. Theory* **41**, 266 (1995).
12. A. D. Wyner and J. Ziv, "Fixed Data Base Version of the Lempel–Ziv Data Compression Algorithm," *IEEE Trans. Info. Theory* **37**, 878–880 (1991).
13. D. K. Gibson and M. D. Graybill, "Apparatus and Method for Very High Data Rate Compression Incorporating Lossless Data Compression and Expansion Utilizing a Hashing Technique," U.S. Patent 5,049,881, 1991.
14. J. P. MacDonald, "File System Support for Delta Compression," Master's Thesis, University of California at Berkeley, May 19, 2000.
15. P. Weiner, "Linear Pattern Matching Algorithms," *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory,* 1973, pp. 1–11.
16. U. Manber and E. W. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM J. Computing* **22**, 935–948 (1993).
17. D. Shapira and J. Storer, "In-Place Differential File Compression," *Proceedings of the Data Compression Conference*, 82–91 (2003).
18. R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM J. Res. & Dev.* **31**, 249–260 (1987).
19. B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM* **13**, 422–426 (1970).
20. A. Tridgell and P. Russell; see *http://rzip.samba.org/*.

Received April 6, 2004; accepted for publication August 8, 2005; Internet publication February 1, 2006

Ramesh C. Agarwal *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (ragarwal@us.ibm.com)*. Dr. Agarwal is an IBM Fellow Emeritus at the Almaden Research Center. He received a B.Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Bombay, and M.S. and Ph.D. degrees from Rice University, where he received the Sigma Xi Award for best Ph.D. thesis in electrical engineering. From 1974 to 1977, he was at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and from 1978 to 1981 he was an Associate Professor at IIT Delhi. Since returning to IBM in 1982, Dr. Agarwal has done research in many areas of engineering, science, and mathematics and has published more than 60 papers in various journals. In 1982, he helped the National Academy of Sciences in the study of the acoustic tapes related to the assassination of President Kennedy, and showed that there is no acoustical evidence for the conspiracy theory. In 1994, he analyzed the floating-point divide flaw in the Pentium chip and showed that for spreadsheet calculations, using decimal numbers, the probability of divide error increases by several orders of magnitude. Dr. Agarwal's main research focus has been in the area of algorithms and architecture for high-performance computing. His current research activities are in the area of data compression and searching through large data sets. Dr. Agarwal has received several prizes and awards, including The President of India Gold Medal. In 1974, he received the Senior Award for Best Paper from the IEEE Acoustics, Speech, and Signal Processing (ASSP) Group. He has received several IBM Outstanding Achievement Awards and an IBM Corporate Award. In 2001, he received a Distinguished Alumnus Award from IIT Bombay. Dr. Agarwal is a Fellow of the IEEE.

Karan Gupta *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (guptaka@us.ibm.com)*. Mr. Gupta is a Senior Software Engineer in the Computer Storage Department at the Almaden Research Center. In 1998 he received a B.E. degree in computer engineering with high honors from the Delhi Institute of Technology, and in 2000 an M.S. degree in computer science from Rensselaer Polytechnic Institute, with a best M.S. Thesis Award. His prior work includes research in traffic engineering algorithms for GMPLS, and his current research interests are in distributed storage systems and data correlation.

Shaili Jain *Division of Engineering and Applied Sciences, Harvard University, Maxwell Dworkin Room 138, 33 Oxford Street, Cambridge, Massachusetts 02138 (shailij@eecs.harvard.edu)*. Ms. Jain is currently a Ph.D. student in computer science at Harvard University, having received an AT&T Labs Fellowship and an NSF Graduate Research Fellowship for her Ph.D. studies. She received a B.S.E. degree in computer science and engineering (summa cum laude) and a B.S. degree in mathematics (with high honors and high distinction) from the University of Michigan in 2004. Her current research interests include randomized algorithms, random processes, coding and information theory, and compression.

Suchitra Amalapurapu *Motorola Inc., 809 Eleventh Avenue, Sunnyvale, California 95089 (asmsuchi@yahoo.com)*. Ms. Amalapurapu received a B.S. degree in electronics and instrumentation and an M.S. degree in information systems from the Birla Institute of Technology and Science (BITS), Pilani, India, in 2000, and an M.S. degree in computer science from the University of Cincinnati in 2003. She is currently a Senior Software Engineer, working on embedded systems.