Date of acceptance 6th November, 2012

Grade ECLA

Instructor

Sasu Tarkoma

Lossless Differential Compression for Synchronizing Arbitrary Single-Dimensional Strings

Jari Karppanen

Helsinki, September 20th, 2012

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet – Faculty Faculty of Science	Laitos – Institution – Department Department of Computer Science

Tekijä – Författare – Author

Jari Karppanen

Työn nimi – Arbetets titel – Title

Lossless Differential Compression for Synchronizing Arbitrary Single-Dimensional Strings

Oppiaine – Läroämne – Subject
Computer Science

Työn laji – Arbetets art – Level Master's thesis	Aika – Datum – Month and year Sep 20th, 2012	Sivumäärä – Sidoantal – Number of pages 93 pages

Tiivistelmä – Referat – Abstract

Differential compression allows expressing a modified document as differences relative to another version of the document. A compressed string requires space relative to amount of changes, irrespective of original document sizes. The purpose of this study was to answer what algorithms are suitable for universal lossless differential compression for synchronizing two arbitrary documents either locally or remotely.

Two main problems in differential compression are finding the differences (differencing), and compactly communicating the differences (encoding). We discussed local differencing algorithms based on subsequence searching, hashtable lookups, suffix searching, and projection. We also discussed probabilistic remote algorithms based on both recursive comparison and characteristic polynomial interpolation of hashes computed from variable-length content-defined substrings. We described various heuristics for approximating optimal algorithms as arbitrary long strings and memory limitations force discarding information. Discussion also included compact delta encoding and in-place reconstruction. We presented results from empirical testing using discussed algorithms.

The conclusions were that multiple algorithms need to be integrated into a hybrid implementation, which heuristically chooses algorithms based on evaluation of the input data. Algorithms based on hashtable lookups are faster on average and require less memory, but algorithms based on suffix searching find least differences. Interpolating characteristic polynomials was found to be too slow for general use. With remote hash comparison, content-defined chunks and recursive comparison can reduce protocol overhead. A differential compressor should be merged with a state-of-art non-differential compressor to enable more compact delta encoding. Input should be processed multiple times to allow constant a space bound without significant reduction in compression efficiency. Compression efficiencly of current popular synchronizers could be improved, as our empiral testing showed that a non-differential compressor produced smaller files without having access to one of the two strings.

ACM Computing Classification System (CCS):
E.4 [Coding and Information Theory]: Data compaction and compression, Error control codes;
F.2.2 [Computation by Abstract Devices]: Nonnumerical Algorithms and Problems---Pattern matching

Avainsanat – Nyckelord – Keywords
synchronization, differential, compression, delta encoding, differencing , dictionary searching, fingerprint, patch, hashing, projective, set reconciliation, content-defined chunking, characteristic polynomial, cpi

Säilytyspaikka – Förvaringställe – Where deposited
Kumpula Science Library, serial number C-2012-

Muita tietoja – Övriga uppgifter – Additional information

Contents

For Nella.

1 Introduction

Often data needs to be *synchronized* with another version of the same data: two separate instances of similar data need to be made identical. After synchronization is complete, one of the instances has been replaced with data identical to the other instance, so that only one version of data remains. For example, a file stored on cloud (such as Dropbox or Google Drive) needs to be updated to a more recent version of the same file that has been modified; or a software application needs to be upgraded to a more recent version of the same application. Often old data is very similar to the new version because only small sections of the data have changed. The two versions of data to be synchronized might be stored on different physical computers, connected by a network. Copying the new version completely requires copying all of the unchanged data, too, even though the unchanged data is already on the destination computer in the outdated version. Time and resources are wasted due to the unnecessary copying. Especially mobile devices utilizing wireless networks are often limited in transfer bandwidth and storage space. As mobile computing is becoming more commonplace, data is being modified on multiple devices and the versions on the devices need to be kept synchronized.

Instead of copying new version of data completely to replace older version of the data, *differential compression* (also called *delta compression*) allows transmitting or storing only the differences between the versions [ABF02]. Differential compressors can substitute sections of data in new version that are already present in the older version with merely references to those identical sections and thus exclude the common content from transmit. An optimal compressor needs to transmit or store only the information that is not present in the other version of data. The intended target version of data can be reconstructed from the other version and the difference information. When differences between data versions are small relative to the total amount of data, differential compression is expected to yield superior compression ratio compared to conventional, non-differential compression. Conventional, non-differential data compressors must store all of the information from data to be able to reconstruct the data during decompression. With differential compression, the amount of data needed to be copied can be just few kilobytes instead of hundreds of megabytes, depending on properties of content.

Differential compression is most suitable for use cases where data transfer bandwidth is limited relative to computational resources, data sets are large relative to transfer bandwidth, and differences between data versions are small relative to total amount of data. The goal is to reduce bandwidth or storage costs [Tri99]. Examples of common use cases include: remote document synchronization, software upgrades (*patches*), incremental backups, cloud computing, human genome research, data deduplication storage systems, and document version control systems [Tri99, ABF02]. Differential compression can also be used to minimize space usage when storing a sequence of versions: for each successive version, only the differences relative to the preceding version are stored (or vice versa). In addition to saving time, smaller data transfers have a lesser probability of failure due to network problems or user impatience [Per06]. Smaller software upgrades increase security and reliability as more patch files can be delivered within given time, which narrows the window of vulnerability [Ada09]. Users may defer or ignore patching because of long download duration, or they may find cloud storage too slow or unreliable to be of practical use.

Conventional, non-differential compression has been studied extensively and there are a number of efficient compressor implementations available. Differential compression as a separate problem has not gained such popularity and there are fewer implementations. This thesis attempts to answer how universal lossless differential compression can be used to synchronize two pieces of arbitrary data, without knowing anything about the data prior to compressing it. We seek to find out whether it is possible to implement a compressor application that is more efficient compared to current publicly available implementations. Main focus is on methods and algorithms suitable for finding the differences and then encoding the differences compactly. We discuss different approaches to differencing, what the main issues in solving the synchronization problem are, and what problems remain to be solved. We also perform empirical testing on the algorithms using a practical implementation to allow choosing the right algorithms for further development of a synchronization tool. We do not discuss application domain specific solutions that are based on known properties of data. For example, lossy compression requires knowledge on how information is allowed to be discarded. We also limit discussing conventional compression, as it is a separate problem, even though it is directly related to performing space-efficient encoding of differences.

This thesis is organized as follows. In chapter 2 we give an overview and describe the general problem of differential compression and provide a framework for later discussion. In chapter 3 we discuss solving the problem of *local differential compression*, where both pieces of data are immediately accessible locally with low temporal access cost. Local differential compression is closely related to conventional, non-differential data compression problem. In chapter 4 we discuss *remote differential compression*, where accessing one of the two strings imposes a high temporal cost due to high read latency or low bandwidth. The two strings are typically stored on separate remote hosts, connected by a slow network. Slow reading restricts a host to reading only the string stored on the same host. Algorithms need to infer which sections of a string have changed without communicating lots of data between hosts. In chapter 5 we empirically compare different algorithm implementations for synchronizing two sets of data both locally and remotely. We provide measurements for compression ratios and algorithm execution times, and test how parameters affect compression. The comparison includes a new implementation based on the research made for this thesis. Finally, in chapter 6 we give the conclusions and summarize the discussion.

2 Differential Compression

In this chapter we introduce the problem of lossless differential compression. We introduce the algorithmic objective, discuss what sub-problems need to be solved, and what are the common issues making solving the problems more difficult. We also discuss how to measure performance so that comparing algorithms in a meaningful way is possible.

In section 1 we discuss what problems needs to be solved to perform synchronization using differential compression and what are the assumptions and conditions for the algorithms discussed in this thesis. In section 2 we describe the sub-problem of finding differences from two arbitrary long strings. In section 3 we describe the sub-problem of encoding and storing the found differences into compressed form, which the user considers as the output from a compression application. In section 4 we describe the sub-problem of reconstructing a copy of the target string from the encoded difference data, when the older version of the string is also present. Finally, in section 5 we discuss different aspects of measuring performance. Deciding which algorithm or set of algorithms is better performing than another is dependent on application domain. Depending on the use case, different solutions and thus different measurements are applicable.

2.1 *Algorithmic Objective*

We first specify some definitions and limitations needed to describe the problem in more detail. Differential compression is done between two strings, a *reference string* R and a *version string* V, that are assumed to have similar content. We specify a string as finite single-dimensional ordered sequence of symbols $S_0 \ldots S_{n-1}$ from some finite constant-size non-empty alphabet Σ. Same alphabet is assumed for both input strings. A substring is a contiguous subsequence of string $[S_i \ldots S_{i+u}) = \{i, u \in \mathbb{N} | 0 \leq i < |S|,\ 1 \leq u < |S| - i\}$ and may overlap other substrings. Symbols are assumed to be constant-width, each requiring $\alpha = \lceil \log_2 |\Sigma| \rceil$ bits of space. Constant-width symbols allow referring to any position of a string in constant time without any special encoding. We denote string positions as zero-based, i.e. S_0 is the first symbol of a string. When giving asymptotic bounds, we denote the length of reference string $|R|$ as n and length of version string $|V|$ as m. In practice, symbols are typically 8-bit bytes and strings are arbitrary files or memory areas: $\Sigma_{file} = \{0, 1, \ldots, 2^8 - 1\}$. When presenting algorithms, we assume strings to be randomly accessible (memory mappable) locally. We give encoding space measurements using bit granularity instead of symbols, because bits as the smallest storage units are better suited for compressed content with variable encoded symbol lengths.

Similarity of two strings means that there are a number of substring pairs of some length between the two strings having zero or close to zero *edit distance*. When two strings are different, there exists a number of finite length ordered sequences of *edit operations* that could be executed on the first string to transform the string into the second string [WaF74]. Each editing operation may be assigned a cost, either static or content-defined [WaF74]. Edit distance measures difference as the minimum cost of edit operations required for the conversion [WaF74]. A *repeat* is a substring that exists in two strings A and B, possibly in different positions, having $[A_p \ldots A_{p+u}) = [B_q \ldots B_{q+u})$; see figure 1. A *left-maximal repeat* is a repeat that cannot be extended towards lower indices, having $A_{p-1} \neq B_{q-1} \vee p = 0 \vee q = 0$. Similarly, a right-maximal repeat cannot be extended towards highed indices, $A_{p+u} \neq B_{q+u} \vee p + u = |A| \vee q + u = |B|$. A *maximal repeat* is a repeat that is both a left-maximal and right-maximal repeat. A *supermaximal repeat* is a maximal repeat that does not overlap with any other repeat. An *intra-repeat* is a repeat where $A = B$. An *approximate repeat* is a substring that can be transformed into a repeat by executing at most d edit operations. Thus, if the edit distance between R and V is less than or equal to d, then V is an approximate repeat of R.

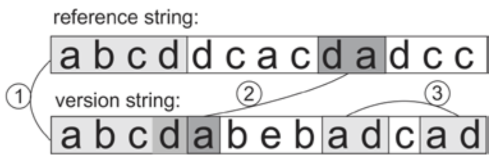


Figure 1: Examples of repeats. A maximal repeat (1), an overlapping repeat (2), and an intra-repeat (3). Note that repeat (1) is not supermaximal within version string, and repeat (2) is not left-maximal.

The algorithmic objective of differential compression is to create a minimum cost delta encoding ΔE from R and V such that it allows reconstructing V into identical V_{copy} when given only R and ΔE [ABF02]. Finding differences between two strings R and V relative to R is the same as finding a set $D = (R \cap V)$ of supermaximal repeats, and then regarding everything not included in the set as differing, $E = V \setminus D$ [ABF02]. For clarity of discussion, we denote the reconstructed version string V_{copy} as *terminus string* in this thesis. Minimum cost for ΔE is defined as requiring as few as possible bits to encode; that is, being shortest possible. Synchronization is performed by copying ΔE to a new location and reconstructing the terminus string V_{copy} in presense of R, after or during which reference string R can be discarded. Multiple strings can be catenated into one and later re-identified and separated by position, which implies an algorithm for compressing multiple strings can be the same as for compressing a single string. Similarly, it is also possible to use multiple reference strings.

The synchronization problem can be divided into three sub-problems: (1) *differencing*, for finding differences between R and V, (2) *delta encoding*, for constructing the bit sequence ΔE that contains the found differences as compactly as possible, and (3) decoding, for reconstructing V_{copy} from the reference string R and delta encoding ΔE [ABF02, SNT04]. From user's perspective, sub-problems (1) and (2) are the *compression* phase and sub-problem (3) is the *decompression* phase. The three sub-problems can be solved separately [You06]. Output from each preceding sub-problem can be considered as input for the following sub-problem and each algorithm could be implemented as a separate program. As we found often done with conventional compression, differencing can be preceded by a preprocessing phase that performs a reversible transform on the strings transforming them into a form that allows finding more repeats. In this case the decoding is followed by a respective postprocessing phase that reverses the transformation. Figure 2 shows the flow of execution.

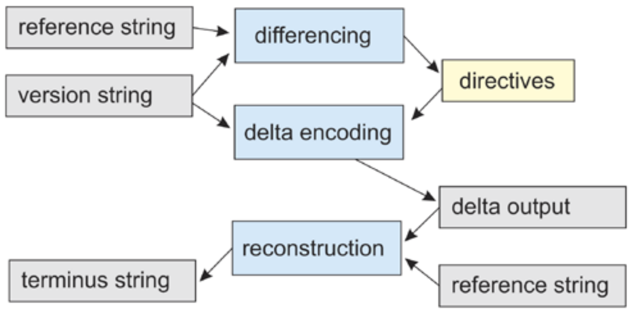


Figure 2: Synchronizing using differential compression.

Differencing and encoding do not require strict sequential processing and the sub-problems can be solved interleaved using concurrent execution threads [BuL97, ABF02]. For example, an encoding algorithm can run in parallel with differencing and start producing output as soon as it receives input, before differencing has completed. Thus, the differencing and encoding algorithms can be chosen independently. We note that some complex and optimized implementations may have implementation-specific dependencies that make more difficult or prevent interchanging the algorithms in practice. Buffering can be used to provide compatibility between algorithms [Tri99, ABF02]. When directive cost is variable, choosing the optimal set of repeats requires knowledge on encoding lengths of the repeats.

To allow remote differencing, execution of the differencing algorithm may be divided to two separate program instances communicating via interprocess communication, each instance having access to only one of the two input strings. The output from encoding algorithm (2) can be stored into a file for later synchronization, or transmitted directly to another host for immediate synchronization. Reconstruction algorithm (3) as the inverse procedure of encoding algorithm is always dependent on the encoding method and must be compatible with the encoding specification [Tri99]. Decoding (3) algorithm is typically run in another process, possibly on another host, and the reference string R and delta encoding ΔE must always be given as input.

In this thesis we discuss algorithms where the encoding and reconstruction algorithms produce lossless compression with high probability of success. The terminus string V_{copy} must be identical to original version string V. However, this does not imply that the differencing should be exact and deterministic, as long as any errors are corrected before committing the terminus string [Per06]. We also limit discussion to algorithms that do not need to have any prior knowledge of the contents of the strings before processing begins, except the strings' lengths. Having prior knowledge could obviously make differencing easier or allow use of content-specific algorithms [LKT06]. Applications could track modifications to strings so that an algorithm already knows which regions of a string contain all the modifications. For example, a source code editor could track which lines have been altered. Similarly, it is known that all the modifications into a log file are insertions beyond the previously known end of the file position. Optimal algorithms with high asymptotic time bounds can be used for short strings. However, additional information is often not available and requiring it would restrict general applicability of an algorithm.

Synchronization using differential compression is not restricted to lossless compression. However, lossy compression requires application domain specific knowledge about how data is allowed to lose information and is thus out of the scope of this thesis. Information loss is often based on some psychoacoustic model, based on knowledge on deficiencies of human senses [ZYR12]. For example, lossy video compression reduces differences in temporally successive pixel color values by using quantization function that maps a larger set of values into a smaller set, thus removing information. Due to removed information, decompression introduces compression artifacts caused by rounding errors from quantization [ZYR12].

Conventional data compression of a string can be viewed as a special case of local differential compression, having a zero-length reference string. There is no separation between local and remote compression because there is no concept of remote host when having only one string on one host. Similarly, local differential compression can be thought of conventionally compressing both the reference string and version string as a single catenated string, but producing an encoded output that excludes symbols from the reference string [TMS02]. Inter-file compression is related to differential compression [You06]. *Solid archives* such as Rar[1] and 7zip[2] are compressed by treating a set of strings as one long catenated string, sharing the encoding contexts, leading to compressed encodings that contain references between strings in the solid archive. All information required for decompression is however still stored within the compressed file and no external reference string is required [You06].

[1] http://www.rarlab.com
[2] http://www.7-zip.org

Even though typically the reference string is considered ‘old’ and version string ‘new’, and the objective is to synchronize by transforming old into new, the roles are interchangeable (synchronize ‘new’ into ‘old’ instead) by simply swapping the denotations. Algorithmically and to clarify discussion, however, in this thesis a reference string is always synchronized into version string. It is up to the application to decide which one of the two strings to compress and which one to regard as the reference string. Document version control systems typically produce the reverse encoding, that is they store the two latest document version completely uncompressed and store the older versions as a sequence of delta encodings. We also do not consider any conflict semantics due to temporal ordering of modifications, such as when both of the strings have been modified and the original reference string no longer exists. Choosing either of the two possible ways to label the strings as reference and version strings would lose information. Such case is an implementation issue and there is no method for autonomically solving the problem. Human interaction or external information on ordering and scope of editing is needed for resolving the conflict [BaP98, RaC01], and synchronization must be done for partial strings. An example of external information is using time vector pairs [CoJ05] to resolve an ordering of synchronizations that loses no information.

Compression in general relies on strings having some encoding redundancy that can be removed to allow re-encoding a string into a shorter form. Compression efficiency is always data dependent [You06]. Usefulness of differential compression relies on the fact that the strings contain maximal repeats [ABF02]. When performing differential compression on two completely unrelated or random strings, there is a high probability that the strings do not have repeats, except those by random chance [ABF02]. When there are no repeats, references to the other string cannot be encoded. Thus, not having any repeats is equivalent to differencing against an empty string, which again is equivalent to not differencing at all and simply compactly encoding the original string. Encoding problem is thus closely related to the problem of conventional, non-differential compression. The two strings do not need to be similar or related at all, it is only necessary to assume that they contain repeats. A compression algorithm must be able to succeed with two unrelated input strings, but it cannot be expected to produce a shorter encoded output when compared to conventional, non-differential compression. When provided with adversarial inputs, a compression algorithm should not produce output that is much longer than the version string [You06], or by extension, longer than conventionally compressed version string.

2.2 Differencing

The two strings to be compressed are believed to be similar, thus certain assumptions are made. It can be expected that the two strings either have many or very few repeats, as the strings either do have similar origins or they don't [Per06]. For example, source codes of two successive versions of the same software are likely to be similar [Per06], but master's theses from different authors are unlikely to contain repeats. Often the repeats are in same order in both strings, because it is not expected that most of the content would have been relocated [BuL97]. When two substrings forming a repeat do not have exact same positions relative to previously found another repeat, the difference between positions is assumed to be small, caused by cumulative offset changes due to short insertions or deletions [ABF02].

Simplest method for finding repeats from two strings is a straightforward symbol-wise comparison. *Hamming distance* measures difference as the number of string positions i having $R_i \neq V_i$. There are however two problems with this approach. First, adding a symbol into or removing a symbol from any position of one string causes an *alignment problem*, where all symbols following the modification position are miss-aligned when compared against the other string [SuM02, ABF02]. Any alignment-altering modification prevents symbol-wise comparison from finding repeats within the shifted range, except if some symbols match by random chance. Adding a single symbol in front of a version string causes the whole string to be detected as different. The alignment problem can be avoided by attempting to find positions where the strings become identical again after finding a mismatching symbol [HuM76]. However, the second problem with symbol-wise comparison is that repeats can be *transposed* within a string [ABF02, You06]. A transposition relocates a symbol into another position, while keeping the string otherwise unmodified. Increasing the comparison position within one string until again reaching a repeat would skip any transposed repeats. Hamming distance overestimates difference and thus symbol-wise comparison is insufficient for finding differences. There is no both trivial and fast method for finding the differences of two strings.

Another type of edit distance, the *Levenshtein distance*, defines three basic edit operations that can be executed on each position within a string. *Change* replaces a symbol with other, without affecting string's length, and is equivalent to edit operation of Hamming distance. *Insertion* adds a symbol into some position of the string, increasing string's length. *Deletion* removes a symbol from some position, decreasing string's length. For example, a string "*abc*" could be transformed into "*bac*" by first removing "*b*" and then re-inserting "b" into the first position, thus using two edit operations. The additional operations solve the alignment problem, but not the transposition problem. The repeats can be located anywhere and in any order within the strings (see figure 3). The repeats may also be overlapping, i.e. one repeat is a partial or complete substring of another different repeat. A search algorithm should find a set of supermaximal repeats having maximum coverage of reference string. Thus, more complex edit operations are needed: a *transpose* operation relocates a symbol into another position. Allowing specifying a count for the transposition operation allows relocating a number of symbols using constant cost [Tic84].

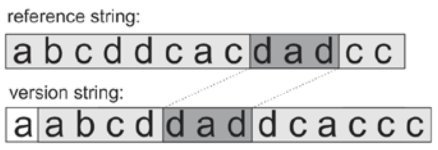


Figure 3: Insertions and deletions (alignment changes) and transpositions make finding repeats more difficult. Only some symbols match in corresponding positions.

One could invent an infinite selection of more complex edit operations that could be used to describe the transformation of a string into another. For example, a *reversal* operation could invert the symbol ordering of a substring, making it possible to reverse all symbols of a string backwards and encode the change using single special instruction. Generic pattern matching would not find any common substrings from the reversed version string, except if it contains palindromes. It would be possible to encode all of the decimal fractions of π using one short mathematical equation, allowing for infinite compression ratio (assuming infinite fractions). The more complex edit operations are allowed, the more extensive the computational requirements for search algorithms are. Having edit operations capable of representing more complex editing would allow finding much shorter edit distance than using the four basic edit operations [You06]. *Kolmogorov complexity* measures the theoretical edit distance that produces one string from another using any possible arbitrary edit operation [Fim02]. The Kolmogorov distance is the size of some function capable of producing terminus string V_{copy} from reference string. It is impossible to calculate the distance programmically, because solving requires (artificial) intelligence [You06]. This thesis focuses on algorithms for finding repeats because other kind of substring matching requires content-specific algorithms or knowledge of the structure of the data [ABF02]. Content-aware pre-processing algorithms can be used to encode strings into a format that allows repeat-finding differencing algorithms to find more repeats [Tri99].

Differencing algorithms need to be able to keep information about one complete string in memory in order to be able to match repeats against any position of the string, which requires $\mathcal{O}(\min(n, m))$ space unless some probabilistic method is used. In practice there is often not enough memory to store all the information from an arbitrary long string and the string must be processed in smaller separate pieces or some of the collected information must be discarded from memory. When an algorithm is allowed to collect only limited amount of information, it cannot make optimal choices nor produce optimal differencing output [ABF02]. An algorithm that had to discard information about a repeat can no longer later match that repeat [ABF02]. When an algorithm is set a constant space bound and is allowed to read strings only once and in sequential order, such as with *stream compression*, no algorithm can find all transposed substrings [ABF02]. Thus, multiple read passes are required for differencing of arbitrarily long strings.

The *atomic unit of change* represents the minimum length of a substring a differencing algorithm can detect as different. When the atomic unit of change used for differencing is larger than the minimum size edit, differencing cannot accurately separate differing data from the unchanged data [Obs87]. Atomic units covering the range of modified symbols may contain some unchanged symbols, in addition to the changed symbols. Within a unit, there can be unchanged symbols either preceeding, following or between the changed symbols. These unchanged symbols are subsequently denoted *slack* [BBG06]. Early implementations in the 1970's used a full line of text as atomic unit of change [Obs87], as increasing atomic unit of change also allows reducing memory space and computation requirements. Changing a single symbol caused the whole line to be treated as different. The more fine-grained the differencing, the more inclusively the repeats can be found, decreasing compression ratio (increasing efficiency) [BBG06]. Atomic unit of change is an issue for remote differencing; larger units need to be used for inferring differences due to need to minimize communication.

A small application-level modification of data could result in a completely different string on the symbol level when an application saves the modified string. Some string post-processing methods also make differencing less effective. For example, encryption attempts to reversibly transform the original string completely so that nothing from the original string is retrievable and any structure or repeating patterns in the string become undetectable [MCM01]. Encryption typically uses preceding encryption output as one of the input parameters to the encryption cipher, so that output for an encrypted block is dependent on the output of preceding block (*cipher-block chaining*). Altering a single symbol will then cause a cascading effect and all following encrypted content after the modification position will also change. Additionally, to prevent encrypted output being similar between multiple executions of encryption on same string, encryption uses randomization such as random initialization vector or "salt" parameter insertion in place of the non-existent preceding output for the very first block. Encrypting the same string multiple times will often produce results that have almost zero common substrings between the encrypted strings. Even if encrypting the same string twice would produce identical results, modifying the original string would produce very different results again because of the cascading effect. Differential compression is almost always ineffective for encrypted content because there are no repeats. The differencing should be done before encrypting the content.

Compressed strings are also problematic to difference. Even though compression does not attempt to scramble the string purposely and the processing is not randomized, the encoding is often done using bit alignment instead of symbol (byte) alignment. Any processing method that modifies a string using atomic unit of change whose size is not integer divisible by the unit used for differencing causes alignment issues [Tri99]. Inserting a single bit into a string causes a cascading alignment change on bit level, causing changes in all following bytes. In addition, compression algorithms use adaptive encoding based on statistics of content and even a small modification could cause a change in the encoding sequences used due to changed statistical ratios [You06]. Attempting to differentially compress already compressed strings usually fails to produce smaller an output because the previous compression process inadvertently removed most of similarity between the two strings [Tri99]. Uncompressing a string, differencing, and then recompressing could circumvent the problem. However, some archives are digitally signed and recompressing is likely to produce a differing archive due to time stamp changes or different compressor version, which causes a signature mismatch [Tri99].

2.3 Delta Encoding

Substituting substrings from version string with references to identical substrings within reference string is an efficient form of compression and produces most of the space savings from differential compression. Assuming that a differencing algorithm produces an optimal output by finding every supermaximal repeat of any length between the two strings, the encoding method will still have a major effect in how much space the delta encoded output requires [You06]. Compressed delta encoding can improve total compression performance significantly relative to uncompressed delta encoding [IMS05]. Differential compression can achieve high compression ratio even without compressed encoding, whereas conventional compression often cannot, unless a string contains mostly intra-repeats. Even a very inefficient human-readable encoding can provide acceptable results in practice when the two strings are similar, because the encoding requires much less space than the referenced substrings. In the optimum case when there are no differences ($R = V$), compression of encoding becomes irrelevant because single reference is enough to encode the whole string. However, compactness of the delta encoding is a major factor in how a differential compressor compares in efficiency relative to other compressor implementations [You06, TMS02].

Further reduction in size of delta encoding is possible by compact encoding of internal metadata, finding intra-repeats within the string, and more compactly encoding the substrings that are to be included into compressed output. Substring compression is done for the repeats that are too short to encode efficiently as references or that the differencing algorithm was unable to find. Directive parameters, such as positions and lengths of referenced substrings, are stored using variable-length encoded integers, instead of using normal fixed-width integers or simple data structures. In practice, compression implies using bit alignment for encoding. The methods are similar to those that are used for compressing strings with conventional, non-differential compression. Thus, additional benefit from compressed encoding is that even if there are not many repeats between the reference string and version string, efficient compression for the delta encoding allows a differential compressor to operate also as a conventional, non-differential compressor.

Both the reference and version strings can contain intra-repeats, in addition to repeats between the two strings. For example, the version string might contain the same substring twice but the reference string does not have any occurrences of the string. The repeats should thus be found from both between the reference and version strings, but also from within those substrings of version string that could not be matched against reference string. Because set of differences is bounded only by the length of version string, both intra-differencing of version string and compression of delta encoding share the same $\mathcal{O}(n)$ space bound and issues of limited memory and processor resources. The intra-repeats should be substituted with references to delta encoding itself, instead of to reference string that does not contain the substrings. For differencing, the already found set of differences can be viewed as a third string, which is differenced against any new differential substring before the substring is appended into the set of differences. Finding repeats from within a reference string does not matter because the reference string will not be compressed and any occurrence of a substring within the reference string is sufficient for referencing purposes.

Delta encoding must be done so that there is no possibility of misinterpreting how the output can be reconstructed when decoding the encoded string. Symbol set used internally in delta encoding ΔE need not be the same as used in R and V because contents of ΔE are only visible to the encoding and decoding algorithms. *Entropy encoding* is a method where size of an encoded symbol is dependent on the statistical probability of the symbol [Ski06]. Each encoded symbol may consist of more than one source symbols. Entropy encoding allows using the shortest tokens for those repeat references and symbols that consume the most space (frequency times length is the largest) and increasingly longer token for more rarely occurring items. Progressively associating longer tokens for content with smaller space requirement leads to longest encoding symbols consuming the least space. Entropy encoding reduces ineffiency of the original encoding. For example, the input strings might only consist of 2^6 distinct symbols even though the strings are originally stored using 2^8 bits per symbol. The set of differences can also contain repeating sequences or patterns of symbols, in addition to the original encoding being inefficient. Compression can reduce the overhead caused by slack when atomic unit of change is long (2^8 ... 2^{11} symbols or more), which is typical in remote differencing [SNT04].

Detailed discussion of general data compression is out of scope for this thesis. For the remainder of this thesis, we denote the general data compression as secondary compression. We briefly describe how compression can be done for delta encoding. Multiple internal symbol sets can be used for different sets of substrings. Because the encoding can be decoded sequentially (random access is not required), variable-size symbols can be used without need for special encoding that require non-zero space overhead. The probabilities for entropy encoding are calculated by a *predictor*, which in the simplest case manages symbol rankings [Mah05]. Multiple different prediction contexts, *context mixing*, can be used to better predict the following symbols and to allow compression requiring less space [Mah05]. For example, a string that contains different type of content in different parts of the string may have very different statistical probabilities for symbols for each part. Detailed description can be found from Mahoney's paper on state-of-the-art encoding methods [Mah05]. Agarwal et al. [AGJ05] suggest monitoring compression efficiency for some initial interval. When compression reduces output size less than some threshold proportion, compression should be stopped to avoid using computation time for non-compressible data, and to avoid producing an output longer than the original.

Entropy encoding can be done with *arithmetic coding* [Ris76] which encodes all symbols into one fractional number between 0.0 and 1.0. When encoding, each parent range is progressively divided into sub-ranges whose sizes are proportional to the statistical probability of symbols being encoded. For example, having 60% probability for "a" and 40% for "b", encoding divides the range into [0…0.6) and [0.6…1] for the first symbol. Range [0…0.36) thus represents "aa" and range [0.36…0.6) represents "ab". For long encoded strings, a range encoder's output efficiency can approach the theoretical limit of information entropy, not counting inefficiencies caused by implementation issues (such as limited machine integer range). *Range encoding* [Mar79] is a form of arithmetic coding where binary integers are used instead of decimal fractions, which allows faster implementation. Arithmetic coding is adaptive and does not require including a coding table into delta encoding, because decoder can reconstruct the same sub-ranges during decoding.

Existing conventional compression algorithm implementations can be used to compress the final delta encoding output [SuM02, MGC07]. Currently one of the best general purpose algorithms is Pavlov's Lempel-Ziv-Markov-chain Algorithm (LZMA) [Igo12, MGC07]. Secondary compression requires as accurate as possible statistical information on the data to be most effective. A more compact encoding is possible if the compression algorithm is modified (optimized) for differential compression [Tri99, SNT04]. Delta encoding includes only the differing substrings, but building of the compression context can refer to the surrounding content that is not included but is available in the reference string. In the worst case when the strings are completely different, the compression problem is exactly the same as with conventional, non-differential compression as there is no surrounding context for the string. Tridgell's [Tri99] testing showed that the *deflate* algorithm produced 1-6% smaller outputs when complemented with a *process but do not emit* operation that reads the preceding input and updates the compression context but does not produce output. For remote differencing, the improvement can be 25% due to inclusion of slack [SNT04].

2.4 Decompressing

Delta decoding algorithm is the inverse operation of delta encoding algorithm and is thus completely dependent on the encoding method. Differential decompression requires both the original uncompressed reference string R and the delta encoded difference data ΔE to be available during decompression. Asymptotic time and space bounds are typically very similar for decoding compared to encoding, because similar or identical data structures need to be constructed. For example, both algorithms need to build coding tables for secondary compression. Decoding algorithms allowed to use temporary space for reconstructing the terminus string do not require the large data structures that are used for differencing, and thus decompression phase requires less memory than compression.

Assuming that contents of V_{copy} need not be accessible externally until reconstruction is complete, an algorithm can iterate through the delta encoding and decode each directive in order [Chr91]. For each directive, the algorithm copies either an encoded substring from the delta encoding itself, or a referred substring from reference string, into some given terminus string position [Chr91]. It does not matter if V_{copy} is in an undefined state while being reconstructed, as it will be identical to original V again when the reconstruction is completed. Any internal structure (such as relational database) is eventually restored when the strings become identical. Synchronization can be performed after decoding is complete by discarding the reference string R and replacing it with the reconstructed string V_{copy}. Reconstructing the version string in some predefined order allows for using the partially reconstructed string even before the reconstruction has been completed fully. For example, backwards linear reconstruction would allow for viewing of latest calendar notes soon after beginning the reconstruction, assuming the latest notes are at the end of the string and that the application supports partial file locking and is aware of the ongoing decompression.

However, storage space is limited and there might not be enough space for a temporary file. For example, mobile devices are often resource limited. Requiring temporary space for terminus string limits the maximum size of a synchronizable string to approximately half of the storage device's capacity. With long strings and short changed sections, copying the identical substrings also require significant time relative to size of changes, especially if the changes are all located near the end of the string. *In-place reconstruction* attempts to solve the problem by modifying the reference string directly, without creating a temporary file [Tic84]. Peak space usage must not exceed $max(m, n)$ [ShS03]. For example, a backup of a log file can be synchronized by just appending the changes without the need to copy existing unmodified content.

In-place reconstruction causes restrictions for the delta encoding and ordering of writing modifications to the reference string [Tic84, BSL02]. By doing the modifications in sequential order, a copy directive in the delta encoding might refer to a position in the reference string which has already been overwritten with content for the terminus string. Reordering the copy directives cannot completely avoid the problem (see figure 4). The conflicting substrings must be temporarily stored in some smaller external storage (of size $w \ll n$) until allowed to be written [BSL02]. The strings can also be kept in memory. We note that when terminus string is longer than reference string ($m \geq n + w$), the extra space w can be used as temporary work space. There is also risk that if for some reason the reconstruction fails or is interrupted, the reference string might end up in an undefined state [Tri99]. Battery failure with a mobile device is not uncommon and leaving an operating system file in an undefined state during firmware update might prevent rebooting the device, thus preventing retrying the synchronization. Length of reconstructed version string can be stored in the delta encoding so that the reconstructor is able to reserve needed space before reconstruction, lessening the probability of a failure due to insufficient space.

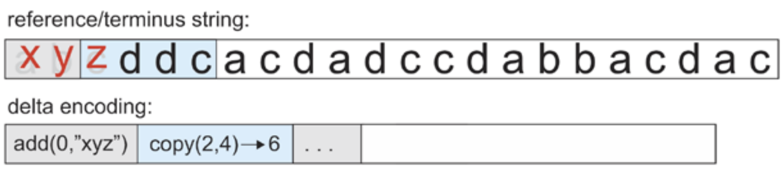

reference/terminus string:
x y z d d c a c d a d c c d a b b a c d a c
delta encoding:
add(0,"xyz")
copy(2,4)→6
. . .

Figure 4: Read-write conflicts prevent in-place reconstruction of terminus string.

To avoid using temporary storage, Burns, Stockmeyer and Long [BSL02] propose modifying the delta encoding to a form that does not cause read-write conflicts. The copy directives are topologically sorted into a *conflicting read-write intervals graph* to minimize number of conflicts. Copy directives that attempt to read an interval another directive writes to are moved forward. A copy directive cannot conflict with itself, as overlapping copying is possible using either left-to-right or right-to-left order. The sorting can be done using standard depth-first topological sort algorithm [CER90]. Not all conflicts can be eliminated by reordering as some lead to cycles in the directed graph. The cycles can be eliminated by converting any of the vertices of a cycle from copy directive into an insert directive (see figure 5). As the sort algorithm puts newly visited vertices on stack, a cycle can be detected by checking if a vertex is already in stack. Converting directives can make the delta encoding longer because more substrings are stored explicitly. A copy directive can be replaced only partially, preserving a reference to non-conflicting interval and converting only the conflicting substring into an insert directive.

Minimizing the increase in length of delta encoding is a NP-hard problem [ShS03], so heuristics is used instead and an algorithm is not guaranteed to minimize the expansion. Burns, Stockmeyer and Long [BSL02] describe two options: an algorithm can select the first one of the cycle vertices it encounters, or attempt to use a greedy method and choose one that locally has the shortest conflicting interval. Choosing the shortest copy operation of a cycle requires more computation and has no asymptotic advantage for worst case bound. Empirical testing showed that choosing the shortest copy results in 0.5% larger and choosing the first copy results in 3.6% larger delta encoding, which indicates the shortest-policy results in less expansion than choosing the vertex that is on top of stack. After removing all of the cycles, the algorithm moves all insert directives to follow copy directives, which avoids the need to include the insert directives into reordering process. Because of the reordering, the directives must be complemented with destination positions in the terminus string, which further worsens the compression ratio. As the intervals of directives are disjoint, any permutation of the directives results in the same terminus string. The algorithm requires $\mathcal{O}(n \log_2 n + \min(n, r^2))$ time when choosing the first vertex, or $\mathcal{O}(r^2)$ time when using local-minima policy (where r is the number of copy directives). The space requirement is $\mathcal{O}(n + \min(n, r^2))$ with both policies. Empirical testing indicated that computation speed has little effect and that reconstruction time is highly dependent on length of delta encoding due to limited write speed capability and non-sequential writing due to reordering [ShS03]. Both cycle-breaking heuristics produce delta encoding with $\mathcal{O}(n)$ space bound [ShS03], but in practice the size increase is small.

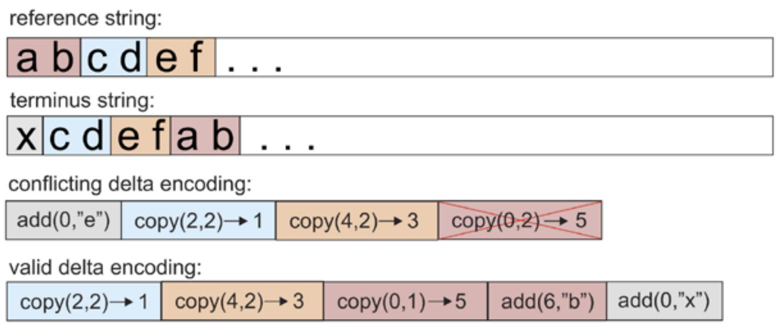


Figure 5: Eliminating read-write conflicts by removing loops.

The reference string used for reconstructing the version string must be identical to the reference string R used when performing differential compression of original version string. Reconstructing the version string using as reference a string that is not identical will result in corrupted outputted terminus string. A separate *digest checksum* can be calculated from the reference string used for compressing and included in the delta encoding, allowing a pre-reconstruction verification check [Tri99]. Patch files are an example where a user might have selected a wrong patch file, not intended for the available reference string. Network transfer is also not free from errors, and long-term storage of a reference string may cause silent data corruption. A checksum enables detecting, but not recovering, from such errors.

2.5 Measuring Performance

When compressing a version string and when the two strings to be compressed contain repeats, differential compression is expected to produce more compact encoding than non-differential compression. Often secondary compression is able to reduce the size of delta encoding considerably and using conventional compression always is sensible [You06, TMS02]. Thus, it is justified to use the length of conventially compressed version string as a benchmark reference for the encoding in all cases. When the strings are completely different, a compressor is expected to match the output size of a non-differential compressor. Thus, a differential compressor should never produce output that is significantly longer than what a conventional compressor produces. Remote differencing always requires a non-zero amount of additional communication overhead for inferring which parts of the strings are different, making it impossible to exactly match a non-differential compressor implementation.

Typically it is assumed that computational resources are plenty compared to storage or transmit resources [Tri99]. An algorithm should have execution time and space bounds such that it can be used with large inputs, otherwise it will be unuseable in practical implementations. In practice this means that an algorithm should have neither polynomial time or space bounds nor linear space bounds with high multiplicative constant [ABF02]. Even linear space bound is often unacceptable, because strings can be much longer than available computer memory can contain. For practical applications, the multiplicative constant in the linear asymptotic time bound is an important factor. Using some algorithm with $\mathcal{O}(n^2)$ time bound for one-time synchronization of large personal documents would require more time than simply copying the documents uncompressed. On the other hand, the same $\mathcal{O}(n^2)$ algorithm might be the best option when producing a patch file for a small software application tool that is about to be transmitted to several millions of customers. The lower transmission cost would amortize the higher computational cost [ABF02].

The three most important measurements that can be used to evaluate the relative performance of algorithms are compression ratio, computation time, and required memory [ABF02, Per06, Tri99]. Compression ratio is measured as the ratio between size of produced delta encoding and size of version string. Smaller ratio equals smaller delta encoding and thus equals better compression. Length of reference string, although required to be present, is not considered. When compressing requires communicating with remote hosts, any protocol overhead should also be included into calculating the size of delta encoding [Tri99]. These three measurements form a trade-off between each other [ABF02]. Increasing computation time allows decreasing (improving) compression ratio, as an algorithm can try more options. Increasing memory requirements allow decreasing compression ratio, as an algorithm has more information to match against. Increasing memory requirement allows decreasing computation, because trying fewer options is countered with more available information. Increasing computation allows reducing memory requirements, respectively.

Algorithmic asymptotic computation bounds ignore implementation issues related to latency. However, different latencies may affect total compression time more than an asymptotic difference of $O(\log n)$ [BSL02]. Random accessing a string, such as out-of-order writing during reconstruction, can become the main factor in execution time [BSL02]. The random access latency with mass storage can be a higher by a factor of thousands compared to sequential access. Even though the asymptotic time bound is not affected, sequential access is always preferred [Tic84, Tri99]. The random accessibility assumption of strings is thus often not valid in practice. A communication round between two remote hosts also incurs a round-trip latency each time. The latency can increase the total time used for synchronization above some other algorithm requiring only a single communication round but that produces less compact encoding without wasting time for low-efficiency communication [Tri99].

Performance and space requirements for compression and decompression are typically asymmetric: decompression does not require the differencing phase or secondary compression, and thus decompression has smaller time and space requirements. For practical applications implementing the algorithms, the objective is often to perform synchronization as fast as possible, which implies a compromise between speed and compression ratio. “Good enough” solutions are often acceptable, because better solutions require more time and space [ABF02]. In general, the goal is to minimize the total synchronization event duration, which may require an application or use-case dependent combination of performance properties. For example, with remote synchronization, processing speed should exceed network bandwidth. The total event duration may include synchronizing multiple string pairs, allowing parallel interleaved synchronization, reducing round-trip latency cost per string. Often it is possible to adjust the operating parameters of algorithms due to the three-way trade-off between compression ratio, execution time and memory requirement [ABF02].

There are also more subtle performance issues, such as capability to start outputting delta encoding immediately while simultaneously still processing the input. When the output can be committed only after the complete delta encoding has been produced, total synchronization time will always include all of the time required for compressing. Small decrease in encoding efficiency could be masked by the ability to start transmitting earlier, essentially eliminating initial latency and converting some of the compressing time to data transfer time. Similarly important is the ability for a client to be able to access a portion of the reconstructed version string while the reconstruction is still in progress. While also an implementation decision, often algorithms prevent accessing the strings until the algorithm is completed.

3 Local Differential Compression

When both the reference string and version string are readable by the same host with low temporal cost, it is possible for a local differential compression algorithm to find all maximal repeats between the two strings by reading both strings completely before producing the delta encoding. Any algorithm iterating through both strings would be extremely slow and inefficient when the strings are stored on separate hosts separated by a slow network with high round-trip latency. It would be faster to simply copy the version string from one host onto another. Optimal differencing requires reading both versions of data completely, and thus simple file copy operations on local file system cannot be made faster using compression: reading both and writing one string will always be slower than simply reading and writing the version string.

Using local differential compression, synchronization of strings can be performed by compressing a version string, transmitting the delta encoded string to the destination host, and then decompressing the delta encoding in presence of reference string. For example, to perform a software upgrade, the author of the software produces a patch file by using the old software version as reference string and new software version as version string. The author can then make the patch file available for downloading. Users can update their old software versions (reference strings) by decompressing the patch file, thus synchronizing their copy of the software with the version available on the server. Similarly, a document editor on a remote host could keep a cached copy of the original document and then produce a delta encoding from the modified version. The delta encoding can then be copied to another host for synchronizing the document with the remote host.

In section 1 we present historical algorithms based on finding a longest common subsequence, leading way to algorithms that most current implementations are based on. In section 2 we discuss an almost optimal algorithm utilizing *dictionary search*, and further discusses modifications to the algorithm to improve time and space bounds to a level that allow practical implementations but lose optimality. In section 3 we discuss differencing using *suffix trees* and other similarly functioning data structures that allow optimally finding matching substrings with $O(n)$ time and space bounds. In section 4 we discuss *projective searching*, where lossy transformation is used to project the compared strings as signals into sub-spaces, and correlation of signals is used to find repeat positions. In section 5 we discuss delta encoding and decoding, where the found differences are stored as compression results, for example into a compressed file, and reconstructed for synchronization. Finally, in section 6 we describe generally applicable methods for tolerating common superfluous differences that make efficient differential compression difficult.

3.1 Subsequence Searching

A common early solution to the differencing problem was finding the longest common subsequence from two strings. A longest common subsequence[3] can be produced by deleting zero or more symbols from either string until they are identical. For example, one possible longest common subsequence for strings "*abcde*" and "*acbed*" is "*abe*". According to Sankoff [San72], solutions to the problem of finding a longest common subsequence were first published in field of speech recognition and bioinformatics, instead of computer science. Needleman and Wunsch [NeW70] solved the problem in 1970 using dynamic programming with $O(mn^2)$ time bound, for comparing protein sequences. Sankoff improved the algorithm's time bound to $O(nm)$. Wagner and Fischer [WaF74] specified the problem of finding the numeric Levenshtein distance as the *string-to-string correction problem* and presented an optimal algorithm for finding the distance. The algorithm could be extended to produce the longest common subsequence, but it has quadratic $O(nm)$ time and space bounds. The algorithm is based on dynamic programming and iteratively calculates successive distances between increasingly longer pairs of prefixes using a matrix. Hirschberg [Hir75] found a way to improve the algorithm's space bound to $O(n + m)$ by not storing results for unnecessary steps. The Hunt-McIlroy [HuM76] algorithm combined ideas from previous algorithms and improved the expected execution time, even though the worst case time bound was $O(mn \log_2 n)$ and space bound $O(mn)$. The algorithm was implemented in Unix *diff* tool to produce a human-readable list of differences from text documents. Implementations considered everything not included in the longest common sequence as differences. Differencing was often used for source code version control and the atomic unit of change was a line of text.

While better than symbol-wise comparison, solutions based on longest common subsequence have the shortcoming that they do not allow finding transposed repeats [Tic84], because a longest common subsequence cannot have out-of-order symbols. For example, in previous example "c" and "d" are still present, they have just changed positions. Finding the longest common subsequence is equivalent to having edit operation set of insert and remove, when constant cost is used for the operations [ShS04]. To solve the transposition problem, Tichy [Tic84] described the problem as *string-to-string correction with block move* and presented an optimal algorithm that was not based on finding a longest common subsequence. Later, the longest common subsequence approach was improved to have $O(min(m,n)d)$ time and space bounds (where d is the number of differences) by both Miller and Myers [MiM85] and independently by Ukkonen [Ukk85]. Their algorithm is based on finding a shortest path in an edit graph [Mye86]. However, since the longest common subsequence method cannot find all repeats, we shall not discuss these algorithms in more detail. When diff-like output is required,

[3] A subsequence does not need to be contiguous.

other algorithms can simulate finding longest common subsequence by excluding output for transposed repeats.

Solving the differencing problem without finding longest common subsequence was attempted by Heckel [Hec78] who presented an algorithm that would however sometimes falsely detect substrings as different. The algorithm first finds lines of text that are unique within a string and can be found from both strings, and infers that the lines must be the same. Range of matching lines is then extended both forward and backward until a differing line is found. When the strings have intra-repeats, the algorithm fails to detect the repeats between strings. For identifying lines Heckel used hashing, which is the fundamendal basis for later differencing algorithms. Heckel also suggested using the hashes for remote differencing.

Tichy's [Tic84] algorithm's idea is to read and compare substrings starting from every position of version string against every position of reference string. For each position, if there is a match, output the longest found maximal repeat as a copy directive, otherwise output an insert directive. Tichy defined two edit operations that both allow combining a sequence of consecutive edit operations of same type into a single operation when the operations target the same substring. The atomic unit of change is a single byte. A *block move* operation is defined as copying a substring from the reference string, whereas an add operation copies a substring from the delta encoding. Block moving allows referencing substrings from another string, regardless of the substrings' relative positions. With these edit operations the problem of differencing is solved by finding all maximal repeats between the two strings, no matter in which positions they are. Each repeat can be substituted with a block move operation, and each difference with an add operation.

Tichy's [Tic84] greedy algorithm operates as follows (see figure 6). The algorithm first starts from the beginning of version string (1) and iterates through every symbol (2). For each position of version string, the algorithm iterates every position of the reference string (3). When matching symbols are found, the algorithm extends comparison further in both strings until non-matching symbols are found or end of reference string is found (10...11). If the found repeat is the first one for the version string position (9), or if the repeat is longer than the previously found repeat (12), the old repeat position is replaced with the new one (13...14). If a repeat was found, its position is output for delta encoding as a copy directive and version string position is advanced by the found substring's length (6). When the end of version string is reached, the algorithm stops. The output now includes all maximal repeats, and their earliest occurrence position in the reference string. A greedy algorithm is such that it in each situation selects the locally optimal choice based on available information, assuming that doing that every time will eventually solve the whole problem optimally. Tichy's algorithm is greedy because it chooses the longest repeat on each step.

```
Algorithm Tichy (R, V)
(1)     v ← 0;
(2)     while (v < V.length)
(3)         FindLongestRepeat(R, V, v, pos, len);
(4)         if (len > 0)
(5)             OutputRepeat(R, pos, len);
(6)         v ← v + max(1, len);

Function FindLongestRepeat (R, V, v, pos, len)
(7)     r ← 0; len ← 0;
(8)     while (r < R.length)
(9)         k ← 0;
(10)        while (v + k < V.length and r + k < R.length and R[r] = V[k])
(11)            k ← k + 1;
(12)        if (k > len)
(13)            len = k; pos = r;
(14)        r ← r + 1;
```

Figure 6: Pseudocode for Tichy's [Tic84] greedy search algorithm.

The algorithm does optimally find all the maximal repeats that the version string and reference strings share [BuL97]. The found set of maximal repeats is however optimal only if constant cost is assumed for copy directives. Alignment changes due to add, remove or transpose operations to version string do not cause issues because the algorithm compares against every possible position. While the algorithm operates within $\Theta(1)$ space bound, the execution time bound is $\Omega(nm)$ and $\mathcal{O}(nm^2)$ which makes the algorithm unsuitable for practical applications because with a long string the required time exceeds the time needed to copy the string. Especially reading through the reference string multiple times would make execution very slow because mass storage is multiple orders of magnitude slower than working memory. Tichy proposed using Knuth-Morris-Pratt pattern matching algorithm for faster substring matching. He identified that the preprocessing required by the pattern search algorithm could be done incrementally one symbol at a time, allowing first searching for one symbol only and then extending the search pattern after each new match.

3.2 Dictionary Searching

To avoid repeatedly searching through the reference string for matching substrings, a hash table based *dictionary* data structure can be used. A dictionary contains positions of short constant-length prefixes from reference string and makes it possible to check if a given prefix is not present without reading through the whole reference string for every check. Having the ability to check for a prefix in $\mathcal{O}(n/q)$ expected time (from a dictionary of size q), it is possible to construct an algorithm that finds all repeats from two strings faster than Tichy's greedy algorithm.

In subsection 1 we present an algorithm based on dictionary searching. The algorithm finds all maximal repeats longer than some small constant number of symbols, but still has $\mathcal{O}(nm)$ time and $\mathcal{O}(n)$ space bounds, even though expected run time is close to $\Omega(mn/q)$. However, memory constraints make it impossible to keep all found substrings in memory, which prevents the algorithm from differencing long strings in practice. Quadratic worst case time bound also prevents compressing long adversarial strings. The following subsections discuss modifications to the algorithm to reduce the memory requirement. Common property of the modifications is limiting the amount of information the algorithm is allowed to store at one time, thus decreasing substring matching capability but allowing operating within nearly $\mathcal{O}(n+m)$ time and $\mathcal{O}(1)$ space bounds. Most practical implementations are based on such modified variation [ABF02]. In subsection 2 we discuss bounding the length of hash chaining in the dictionary due to hash collisions. We also discuss repairing the damage by keeping a history buffer and using multiple passes through the strings. In subsection 3 we discuss restricting visibility of an algorithm by differencing within a sliding window instead of complete string, thus requiring memory in proportion to window's size instead of strings length. In subsection 4 we discuss differencing a sparse subset of substrings, requiring less memory in proportion to sparsity. A sparse differencing algorithm assumes that the repeats are long and that ignoring some intervals allows still finding partial repeats, which can then be extended to maximal.

3.2.1 Almost Optimal Greedy Algorithm

Obst [Obs87] presented an improvement to Tichy's algorithm by using *shingling* [You06] to search for variable-length substrings more efficiently. A string is viewed as contiguous sequence of overlapping short fixed-length *seed prefixes* of length p (*p-grams*) starting from every successive position (see figure 7). *Fingerprint* hash values are calculated for every seed prefix of the string and for each seed prefix of the search patterns. Each fingerprint maps a 2^{8p} bit seed prefix into a 2^q bit hash, such that $q \leq 8p$ [You06]. Obst's version of the greedy algorithm is essentially a multiple-pattern version of Rabin-Karp pattern matching algorithm [KaR87, BeM99]. Obst's algorithm solves the exact string matching problem with multiple search patterns, where the search patterns are all possible substring prefixes of length p from version string.

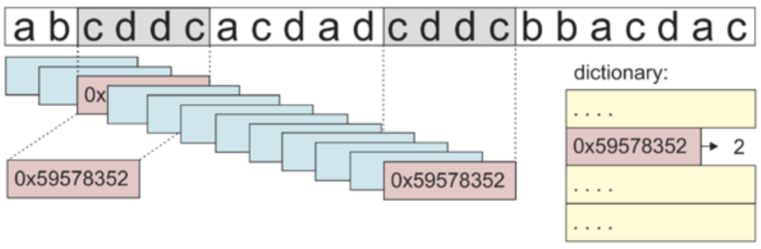
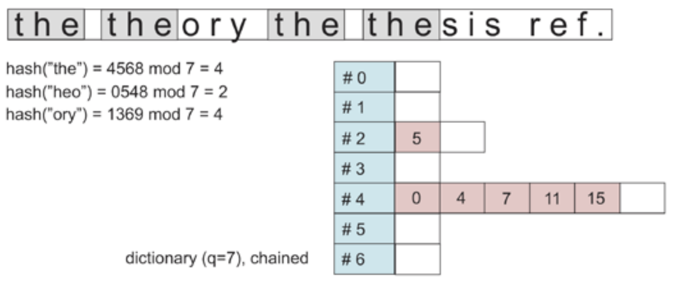


Figure 7: Shingling is used to compute fingerprints into a dictionary data structure.

Rabin-Karp pattern searching algorithm [KaR87] calculates a hash value from the search pattern and then iterates through every position of the string, calculating a corresponding hash value for each overlapping substring of the same length as the search pattern. Hash value from each position is compared against the search hash value. A hash function must always produce the same output for the same input substring, so a failed comparison due to differing hash values is always a reliable result, but a successful comparison might be a *false match*. When the hash values do not match, substrings cannot be identical and the algorithm proceeds to the next string position. Succesful hash comparisons are verified using a symbol-wise substring comparison, to filter out the false matches. One pattern can be searched from a string in $\mathcal{O}(np + h)$ worst case and $\mathcal{O}(n + h)$ expected time [KaR87], where p is the length of search pattern, and h is the time required to calculate a single hash value.

A dictionary is a hash table data structure that uses fingerprints as keys and stores string positions of reference string corresponding to each fingerprint, essentially mapping fingerprints to substrings (see figure 8). Fingerprints from version string can be used to query reference string positions from the dictionary. Zero results indicate with certainty that the substring cannot be found from reference string, without the need to iterate through reference string. A hash function may produce identical results for multiple string positions, because the same prefix can occur in multiple times in different positions. Multiple different prefixes may also produce the same hash value due to *hash collision*, when the hash function inadvertently outputs identical values for the different prefixes. The dictionary data structure must thus be able to store multiple string positions for each fingerprint by implementing *hash chaining*. Each chain entry must be verified separately, requiring $\Omega(c)$ symbol-wise comparisons (where c is the number of collisions). Constant-time comparison of prefixes is thus only possible when a dictionary is uniformly populated and each entry contains only a small constant maximum number of prefix positions [ABF02]. Worst case input contains only a long sequence of same symbol, causing all fingerprints to be mapped into same hash table chain, thus causing the dictionary to act like a list with $\mathcal{O}(n)$ time bound for lookups [ABF02]. Expected query time is $\mathcal{O}(n/q)$ with random content and perfectly uniform hash function (where q is the maximum size of dictionary). Because in practice content is not random and hash functions are not perfect, the query time is higher. A size-unrestricted dictionary requires $\mathcal{O}(n - p + 1) = \mathcal{O}(n)$ space [ABF02].



t h e t h e o r y t h e t h e s i s r e f .
hash("the") = 4568 mod 7 = 4
hash("heo") = 0548 mod 7 = 2
hash("ory") = 1369 mod 7 = 4
dictionary (q=7), chained
# 0
# 1
# 2
5
# 3
# 4
0
4
7
11
15
# 5
# 6

Figure 8: A dictionary data structure maps fingerprint values (hashes) into position information.

Obst's [Obs87] algorithm is similar to Tichy's algorithm, but instead of searching through the whole reference string for each position in version string, it compares the hash values from a dictionary. The algorithm executes as follows (see figure 9). Initially the dictionary will be empty (1). The reference string is first iterated through and for every position a fingerprint of each respective substring prefix of length p from the string is calculated (2…6). Calculating the fingerprints for every position thus requires $\mathcal{O}(nh)$ time. After the reference string has been iterated through, there are $\Theta(n-p+1)$ positions stored in the hash table. The version string is then iterated through and a fingerprint is calculated for each position, the same way it was done for the reference string, requiring $\mathcal{O}(mh)$ time (7…8). Each fingerprint is queried for from the dictionary (9). When there is no matching entry in the dictionary, the algorithm can be sure that the seed prefix cannot be found anywhere from the reference string. Thus, reference string cannot contain any substring that has the same prefix. When one or more matching fingerprints are found, each must be verified using symbol-wise comparisons (11…12), requiring $\mathcal{O}(p)$ time per dictionary chain entry. When a prefix is verified to be identical, the prefix is a repeat and comparison is extended forwards in both strings until a differing symbol is detected. All dictionary chain entries are always checked and longest matching substring is chosen for delta encoding as a copy directive (16…17). If symbol-wise comparison of prefix fails, the algorithm proceeds as if hash values would not have matched (18). When no matching prefixes are found, the symbol from version string's iterator position is outputted as a differing symbol to the encoding algorithm. Multiple sequential differences can be combined into one insert directive. If a match was found, iterator position in version string is moved to the next symbol following the matching substring, to avoid overlapping substring references. A single-pass algorithm can be considered to have $\mathcal{O}(n)$ time bound even if it makes a limited number of random access reads [ABF02]. We note the algorithm can be modified for approximate pattern matching by allowing some non-matching symbols when extending the verified match.

```
Algorithm Obst (R, V, p)
    dict ← new Dictionary;
    r ← 0; v ← 0;
    while (r < R.length - p)
        fp ← Hash(R, r, p);
        dict.Insert(fp, r);
        r ← r + 1;
    while (v < V.length)
        fp ← Hash(V, v, p);
        chain ← dict.Query(fp);
        k ← 0; len ← 0; pos ← 0;
        while (k < chain.length)
            test ← VerifyAndExtendRepeat(R, V, chain[k].pos, v);
            if (test > len)
                len = test; pos = chain[k].pos;
            k ← k + 1;
        if (len > 0)
            OutputRepeat(R, pos, len);
        else
            OutputDifference(V, v);
        v ← v + max(1, len);

Function VerifyAndExtendRepeat (R, V, r, v)
    c ← 0;
    while (r + c < R.length and v + c < V.length and R[r+c] = V[v+c])
        c ← c + 1;
    return c;
```

Figure 9: Pseudocode for Obst's [Obs87] differencing algorithm.

All fingerprints from reference string must be stored in the dictionary, so the space bound has worsened from $\mathcal{O}(1)$ to $\mathcal{O}(n)$, when compared to Tichy's algorithm. None of the search patterns can be shorter than the seed prefix length, because fingerprinting function's input length sets a lower bound for the length of matchable prefixes. No repeats shorter than the prefix will be found and the shorter repeats will be encoded as differences. The algorithm is thus optimal for only strings containing only repeats that are as long as or longer than the chosen prefix seed length. Ability to find only longer matches is not necessarily a problem, depending on content and on the efficiency of encoding method. In practice, encoding schemes must ignore short repeats whose encoding would require more space than encoding the repeats as extensions into neighboring insert directives. The seed prefix length can be given as a parameter. Choosing shorter prefix length will cause false matches more often, causing more symbol-wise substring comparisons that will not match. Optimal prefix length is dependent the delta encoding algorithm and string content: the average encoded reference must be shorter than encoding the repeats as additions [You06]. For naive encoding schemes, optimal seed length is between 12 and 18 bytes, depending on content [Chr91, ABF02]. For state-of-the art encoding schemes such as used in LZMA compression algorithm [Igo12], the optimal seed length is between 2 and 4 bytes, depending on string length (longer strings require more bits for position information).

A good hash function must be fast and produce as uniform as possible distribution of hash values to prevent unevenly populating a hash table. If the hash function produces a non-uniform distribution of values or the set hash of possible hash values is too small, many strings will have the same hash value, which will cause large number of verification comparisons [ABF02]. Verification comparison requires random accessing the reference string, which can be slow if the prefix positions are spread wide, causing disk cache misses. Especially a set of strings that can be permutated from one string should not cause the same hash values to be calculated because such cases are common in practice [SNT04]. When strings are similar, both will have similar fingerprint collision distribution, meaning that the longer the chains, the more often the longest chains have to be checked. Repetitive patterns are thus problematic, because each occurrence will have the same fingerprint.

Calculating a separate hash value for every seed prefix would require including each string position $p-1$ times into hash calculation, due to overlapping. However, it is possible to use a simple and relatively fast *rolling hash function* [Rab81, ABF02]. After initializing the hash function and calculating the initial hash value $h(S_i)$, the next hash value $h(S_{i+1})$ can be calculated from $h(S_i)$ in constant time independent of input length. Rolling hash functions calculate hash values from a *sliding input window*: each time the string input position is advanced, the symbol that is dropping out of the window's range is subtracted from the hash value and the next incoming symbol is calculated into the hash value. On each step one symbol is dropped and one symbol is added. Preceding content thus does not affect the hash value and two identical seed prefixes produce identical hash values. Reichenberger [Chr91] compared different hash functions for differencing and identified *Rabin fingerprinting* [Rab81] suitable due to ability to calculate fast and provide uniform enough distribution of hash values. Another commonly used fast hash function is Adler-32, which is a variation of Fletcher checksum [Tri99]. Reichenberger inferred that using longer prefixes for hashing produces a more uniform distribution of hash values.

Rabin fingerprinting function is defined by choosing an irreducible polynomial and degree of polynomial (polynomial coefficients for each successive symbol) and an initialization vector representing empty string (that can be viewed as a prefix for any input string) [You06, appendix A]. The function is defined as $h(x) = S(x)\ mod\ q$, where q is the irreducible polynomial of degree k over the finite field $GF(2^a)$, where $a = 1$ [You06]. For calculating the hash, the string input is viewed as a large number $f(x) = s_0 + s_1x + \ldots + s_{n-1}x^{n-1}$ and reduced modulo polynomial q to acquire a *residual* [You06]. The operation is not the normal arithmetic modulo. The irreducible polynomial is stored as a bit string of $k+1$ bits where each bit represents a coefficient of $h(x)$ [You06]. There are $2^k/k$ irreducible polynomials to choose from (2^{27} for 32 bits) [You06]. There are only $1/\log(k)$ prime numbers while there are 1/k irreducible polynomials [You06]. Initialization vector is required to produce different results for polynomial $S(x) = 0$ of different lengths (a string consisting of only zero bits) [You06]. Pseudo-random generator is used to generate the polynomial, which is then tested for irreducibility [You06]. This allows specifying new hashing functions that produce different hash values for an input than some other hashing function using different polynomial. The function can be tested empirically and the polynomial chosen based on test results [Tri99].

When hash value range is $0 \ldots q$, the expected number of hash collisions using Obst's algorithm is $\mathcal{O}((n - p)/q) = \mathcal{O}(n/q)$. Worst case time bound with adversarial inputs is now proportional to the number of matching checksums and is $\mathcal{O}((n - p + 1)m)$. For completely uncorrelated or identical strings, expected execution time is $\mathcal{O}(n + m)$. In general case, the expected time is $\mathcal{O}(m + mn/q) = \mathcal{O}(m)$, when $q = n$. In practice Obst's algorithm often does not achieve the expected execution time because structured strings have sequences of repeated symbols (such as zero padding) that cause hash collisions. Multiple hash collisions then accumulate into a single hash chain [TMS02]. When a long chain is encountered, a differencing algorithm could test if the next string offset hashes into a shorter chain, and use shorter chain instead to reduce worst-case behaviour [TMS02].

Hash value range needs also to be small enough to allow comparing two hash values efficiently. Hash values can be scaled to a given value range by using modulo arithmetic. In practical implementations the modulus q is usually chosen to be a prime number in order to avoid any repeating structure in the string. The most trivial hash function would be to interpret the symbols in the prefix as base-n integer (directly interpreting the bits as an integer, instead of symbols). Calculating the hash would be fast, but probability of uniform distribution would be low. For example, most bytes in an English-language text document have zero as the most significant bit, causing some of the hash functions output range to be left always unused. Searching for an entry in the dictionary can be made faster by using a small hash prefix table that contains partial hash values [AAJ04]. When checking for if a given hash value is stored in the dictionary, the prefix table is first checked for the partial hash value [AAJ04]. Another similar technique is Bloom filters [TRL12], which support probabilistic set membership queries for the fingerprints. A Bloom filter is a composite mask of bits from fingerprints of the member set and may return a false positive due to more bits being set as the number of members grows. Negative member queries however are reliable and allow skipping queries from the dictionary.

There are other fast string searching algorithms, such as Knuth-Morris-Pratt algorithm and Boyer-Moore algorithm that are more efficient than Rabin-Karp for searching a single string. These algorithms avoid comparing some of the symbols by requiring preprocessing for each search pattern. After performing a comparison, these algorithms infer the number of symbols that can be skipped because they cannot match the pattern. They cannot infer the skip distance when there is arbitrary number of search patterns. With multiple-pattern searching the preprocessing required by the algorithms needs more calculations than Rabin-Karp, which does not attempt to skip symbols.

3.2.2 Bounded Hash Chaining

Main problems of Obst's greedy dictionary search algorithm are $\mathcal{O}(nm)$ worst case time bound and $\mathcal{O}(n)$ space bound. The time bound is problematic because in practice hash chains can become long with arbitrarily long strings [ABF02]. Linear space bound is worse than Tichy's greedy algorithm's constant space bound. Improving the asymptotic time and space bounds of a search algorithm while still using the fingerprinting approach can be done by limiting or discarding collected information. Reducing available information will also prevent an algorithm from optimally finding all matching substrings [ABF02].

Trendafilov et al. [TMS02] describe a restricted hash chaining policy, where collisions due to identical seeds within the reference string are not appended into the collision chain. Let's assume that during fingerprinting of a reference string, the algorithm encounters a collision leading to an identical earlier found repeat at position i and of length k, and the latter found repeat could be matched longer than the earlier. Instead of adding the position of a duplicate repeat into dictionary, the algorithm jumps $k-(p-1)$ positions to the end of the repeat and continues fingerprinting. No fingerprints from substring $[S_i \ldots S_{i+k-p})$ are inserted. Each hash chain can contain more than one position, but only from unique seed prefixes that caused a hash collision. If the latter found repeat is not longer, the earlier repeat can be used to encode the copy directive. When differencing against version string and when the algorithm encounters a repeat, it can jump $k-(p-1)$ positions and check if that leads to another repeat that extends the repeat further than where the first fingerprint match lead to (see figure 10). This improvement limits worst case behavior and uses memory more efficiently, but dictionary still has $\mathcal{O}(n)$ space bound.

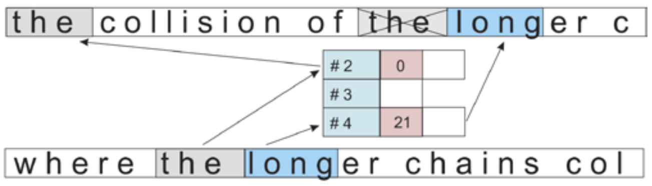


Figure 10: Cascading position jumps for colliding hash values.

An asymptotically effective space and time limitation is restricting the chain length per dictionary entry to some small constant number c, instead of storing all found seed prefixes that cause fingerprint collisions [BuL97, ABF02, ChM06]. This is equivalent to comparing against only some c of the stored positions of a fingerprint in the optimal greedy algorithm. Having only a constant number of fingerprints to compare per string position lowers the time bound from $\mathcal{O}(nm)$ to $\mathcal{O}(mc+q)$, where q is the size of dictionary (a constant typically in the range of 2^{16} to 2^{24}). The space requirement is decreased to $\mathcal{O}(qc)=\mathcal{O}(1)$ if only prefix positions and match lengths are stored in the dictionary.

Burns and Long [BuL97] and Ajtai et al. [ABF02] note that the change will cause degradation of output compression ratio in any adversarial inputs because large number of positions that would match are not stored in the dictionary. Substring search (matching version string to reference string) will have limited visibility and the algorithm will no longer be optimal in finding matches. However, if one position cannot find a match due to collision, some other position might, because any seed prefix within a repeat is sufficient to locate a repeat [Mac99]. Ajtai et al. found that a restricted algorithm finds more differences by a factor of 2 to 4 relative to optimal differencing, on typical input. Dictionary size directly affects compression performance: the larger the dictionary, the more potential substring positions will be available for comparison. Hash function's quality affects the dictionary's fill ratio: the more there are hash collisions due to a poor hash function, the more of the dictionary remains empty. With unlimited chaining, hash collisions make the Obst algorithm slower, but with limited chaining the collisions prevent matching the rejected repeats. This is equivalent to having a smaller dictionary size when considering repeat matching capability. Dictionary size should be increased linearly relative to length of reference string or there will not be enough dictionary entries for same number of fingerprints per equal-size subsection of a string, leading to decreasing probability of finding matches as strings become longer.

Number of fingerprints calculated from the strings will still be the same, so when a fingerprint collision occurs, some heuristics is needed to decide which one of the two positions with colliding fingerprints to keep stored in the dictionary and which one to ignore. Limiting the number of stored colliding positions can cause *blocking* of the other different substrings with the same seed prefix and fingerprint [ABF02]. When there is a fingerprint collision, the existing stored fingerprint forces the algorithm to encode the substring as an insert directive [ABF02]. Even if there is a fingerprint leading to a matching substring, it might not be the longest possible match because the longer match has been blocked from being inserted to the dictionary, or the longest match has not yet been found [ABF02]. Search algorithm might later find a match for the remaining suffix of the matching substring, but delta encoding the complete reference will then require two or more directives instead of just one [ABF02].

Ajtai et al. [ABF02] suggest different *collision policies*; one is to always reject colliding fingerprints and thus store only the first position for each fingerprint. We denote this as the *keep-first policy*. MacDonald [Mac99] suggests using pseudo-random Boolean function to simulate extra bytes for the fingerprint for deciding whether to reject or replace the fingerprint, and to preserve uniform distribution of fingerprints. A counter variable can be used to give the replacement a probability of $1/(c+1)$, where c is the number of earlier collisions for the dictionary entry [Mac99]. Either way, only one fingerprint can be stored per dictionary entry and the dictionary does not implement chaining. With this policy, a reference string is fingerprinted completely before iterating through a version string, just as with Obst's algorithm. This way the dictionary will contain all of the fingerprints that are possible to be stored from reference string, which makes it possible to match substrings from latter part of reference string against early part of version string. Because large number of seed prefixes cannot be matched, repeat extension by symbol-wise comparison is done backward in addition to forward, in case the matched fingerprint is not for the prefix of a repeat. With extending to both directions, it is sufficient that the fingerprinted seed prefix is anywhere within a repeat.

The backward comparison modification causes algorithm's time bound to become super-linear instead of $O(n)$ [ABF02]. When the algorithm finds a match, it can jump forward for the length of the match, but it could miss some overlapping longer matches. Another issue is that backward-extended references may overlap with earlier already encoded output [ABF02]. When processing large files and keeping the dictionary size constant, the dictionary will eventually become full and the algorithm will only be able to make comparisons against the fingerprints found from earlier part of the reference string [ABF02]. The longer the reference string or smaller the dictionary, the more likely these stored positions are to be localized to beginning of the reference string [ABF02]. As the assumption is that matching substrings between strings are positionally close to each other, not being able to compare against substrings within the latter part of the reference string becomes the bigger problem the longer the reference string is [ABF02].

To mitigate the decrease in capability to match repeats, Ajtai et al. [ABF02] suggest that corrective buffering can be used to increase number of found matching substrings and reduce the number of delta encoding directives by retroactively changing recent output decisions before committing them to delta encoding. *Encoding correction* can be implemented with a *lookback buffer* containing a number of recent encoding directives (see figure 11). The buffer must be both modifiable and searchable. The lookback buffer can be implemented in various ways. Tail correction requires inspecting the last stored directive on every iteration. A simple fixed-size buffer with "first in, first out" semantics allows fast access to the last directive. Within the buffer, simplified temporary encoding directives can be used. *Tail correction* can be used to check if the last directive inserted into the buffer overlaps with the currently found match. When the last directive is a reference and can be completely merged into the current directive, the last directive is combined with the current. When the last directive is an insert directive that partially or completely overlaps with the current copy directive (due to backwards extending), the overlapping suffix of insert is removed. For example, consider a substring "abcabcdefg": if the algorithm has just found a repeat position for the first "abc", but then later finds a repeat anywhere in the remaining longer substring, the algorithm can replace the encoded copy directive with the longer one.

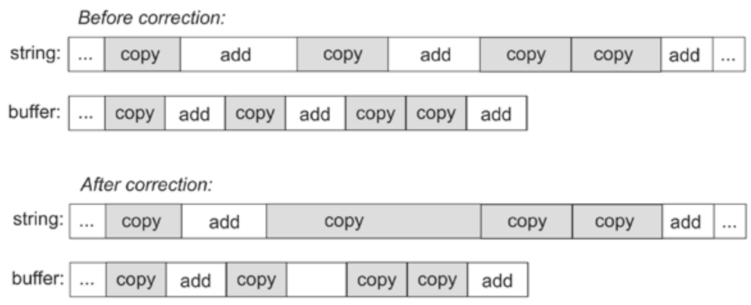

Before correction:
string:
... copy add copy add copy copy add ...
buffer:
... copy add copy add copy copy add
After correction:
string:
... copy add copy copy copy add ...
buffer:
... copy add copy copy copy add

Figure 11: Correcting the previous choices with better choices by using a history buffer.

Another collision policy suggested by Burns [BuL97] is to clear the dictionary entry after finding a repeat, denoted as the *flushing policy*. This avoids the problem of dictionary becoming read-only on long strings. Because the respective positions of matching substrings are assumed to be nearby each other, a further assumption can be made that after some differing symbols, the strings would again become identical and search will find a pair of *synchronized offsets*. Reference string is not fingerprinted completely before differencing against version string, but instead both strings are processed in parallel. On each step, fingerprints are calculated from seed prefixes of both strings. When a string position from other string is already found from the dictionary, a possible repeat has been found and the verification comparison is done; otherwise the collision is ignored. The dictionary does not need to implement chaining and thus the space bound is $\mathcal{O}(q) = \mathcal{O}(1)$. After a repeat is found, the dictionary entry is cleared and the algorithm resumes hashing as if the strings started from current position and the previously processed data had never existed. None of the substrings already fingerprinted can be matched later. Similarly, if a repeat is located further in strings and it has not yet been found, it cannot be matched. Another issue is that if there is a shorter repeat preceding a synchronized offset, the algorithm assumes the offsets are nonetheless synchronized and outputs a copy directive for the short repeat. The algorithm might later match the latter part of the longer repeat, but requires another copy directive to encode it. When the strings are not transposed and version string contains only insertion and deletion edits, flush policy approach will approximate the optimal greedy algorithm's search ratio closely [ABF02]. Time bound of the algorithm is $\mathcal{O}((n+m)p\ +q)$ [BuL97].

Ajtai et al. [ABF02] presented an improved version of the flushing algorithm that does not clear the dictionary when a repeat is found. We denote this as the *replace policy*. Instead of ignoring fingerprint collisions, the earlier position is replaced with most recent position corresponding to a fingerprint. Instead of becoming read-only, the dictionary is updated even when full. Chedid and Mouawad [ChM06] propose storing k latest fingerprints using a circular linked list. Each time a fingerprint collision occurs and when there already are k positions stored, the smallest position is removed from the collision chain. Storing some past fingerprints in the dictionary allows matching some spatially close transpositions. Repeats are extended both backwards and forward, again causing super-linear worst case time bound.

Ajtai et al. also extended the correction: *General correction* searches the whole lookback buffer, instead of checking the last item (see figure 12). Directives can be corrected multiple times. The oldest directives are committed when the buffer becomes full. Ajtai et al. note that encoding correction improves the capability of finding transposed substrings when the substrings are close enough each other so that the directive related to previously found substring is still stored in the buffer. If the lookback buffer is allowed an unlimited size and dictionary stores the latest fingerprint, instead of blocking fingerprint collisions, an algorithm can find all transposed strings. General correction also makes it possible to replace multi-part references caused by “spurious” repeats with longer single references that are found later. When processing the version string linearly, the string offsets in the buffer are in increasing order. There are no duplicate positions in the buffer. These two properties allow using binary search within a buffer. Simple buffer does not allow fast insertion of new directives, which prevents some corrections from being done; a balanced binary tree can be used instead. Noop directives can be used in place of removed directives. Since each iteration may now require a search in the buffer, the time bound of the whole algorithm becomes $O(n \log_2 b)$, where b is the size of the buffer.

```
Algorithm DictSearchWithCorrection (R, V, p)
(1)   dictR ← new Dictionary;
(2)   dictV ← new Dictionary;
(3)   r ← 0; v ← 0; vs ← 0;
(4)   while (v < V.length)
(5)       fv ← Hash(V, v, p);
(6)       dictV.Insert(fv, v);
(7)       while (r < R.length - p)
(8)           fr ← Hash(R, r, p);
(9)           dictR.Insert(fr, r);
(10)          posR ← dictV.Query(fr);
(11)          posV ← dictR.Query(fv);
(12)          if (posR != null)
(13)              len ← VerifyAndExtendRepeat(R, V, rm ← posR, vm ← v);
(14)          else if (posR != null)
(15)              len ← VerifyAndExtendRepeat(R, V, rm ← r, vm ← posV);
(16)          if (len > 0)
(17)              CorrectingEncode(R,V,r,v,rm,vm,len);
(18)      else
(19)          OutputDifference(V, v);
(20)      r ← max(rm + len, r + 1);
(21)      v ← max(vm + len, v + 1);

Function VerifyAndExtendRepeat (R, V, rm, vm)
(22)  len ← 0;
(23)  while (rm + len < R.length and vm + len < V.length and R[rm+len] = V[vm+len])
(24)      len ← len + 1;
(25)  while (rm - 1 >= 0 and vm - 1 >= 0 and R[rm-1] = V[vm-1])
(26)      rm ← rm - 1; vm ← vm - 1;
(27)      len ← len + 1;
(28)  return len;

Function CorrectingEncode (R, V, r, v, rm, vm, len)
(29)  if (vs < vm)
(30)      buffer.append(OutputDifference(V, vs, vm-vs));
(31)      buffer.append(OutputReference(R, rm, len));
(32)      vs ← vm + len;
(33)  if (vm < vs < vm + len)
(34)      CorrectTail();
(35)      buffer.append(OutputReference(R, rm, len));
(36)      vs ← vm + len;
(37)  if (vm + len < vs)
(38)      CorrentGeneral();
```

Figure 12: Pseudocode for one-pass chain-limited dictionary searching with general correction.

Ajtai et al.'s [ABF02] empirical testing showed that for dictionary of equal size, comparing to flush and replace policies, the keep-first policy produces better matching ratio for short strings due to having found longer matches from latter part of the reference string, but is also slower and loses matching ratio as the strings become longer. Ajtai et al. conclude that none of the policies is clearly better for short strings, but with long strings the keep-first policy prevents finding repeats and thus the replace policy is preferred. Ajtai et al. also suggest using two dictionaries, one for each string, to allow storing separate positions for two colliding fingerprints that do not correspond to a repeat. We note that more complex heuristics could be used, such as using a cost function to evict least referred fingerprints when using limited-length chaining, but having to store counter variables would make the dictionary larger.

3.2.3 Sliding Window

To attempt to solve the problem of matching transposed strings while avoiding the filling of dictionary, Trendafilov, Memon and Suel [TMS02] suggest using a *sliding window* approach. This method is based on the assumption of spatial locality: repeats are assumed to have relatively close positions, even when transpositioned. Thus, choosing some large enough arbitrary length w for the window, the assumption is that corresponding repeats from both strings are contained within the sliding window. Fingerprinting and subsequent differencing is done only within the window. The reference string is fingerprinted one window at a time, as if the window was the whole reference string (see figure 13).

When version string fingerprinting position has advanced enough, some heuristics function will trigger window advancement. For example, weighed average of recently found repeat positions (by copy length) can be used as a threshold [TMS02]. The advancement function will clear the dictionary and increase window's starting position by some number of symbols based on heuristics, e.g., half the window's size or some number of symbols based on distribution of fingerprints stored in the dictionary. After sliding the window, the algorithm is restarted from the new window position. Not being dependent on lengths of either string, the sliding window reduces algorithm's time and space bounds to $O(nw)$ [TMS02].

The sliding window method is compatible with both the Obst greedy algorithm and the chain-limited modified versions of the algorithm. When combined with the greedy method, an algorithm will optimally find all repeats from within the sliding windows' range only, and cannot find any repeats crossing or outside the sliding window boundary [BeM99]. When used with chain-limited dictionary and keep-first eviction policy, the sliding window allows avoiding overflowing the dictionary and gains ability to match more latter repeats with the expense of losing ability to match against early repeats .

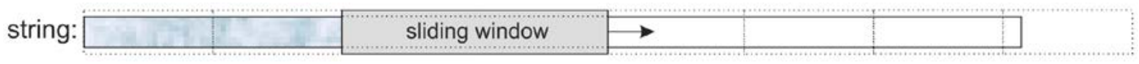


Figure 13: Sliding window for fingerprint dictionary.

Sliding window's size can be chosen based on the amount of memory available or based on empirically predetermined study on the effect on compression ratio using typical input strings. Window size can also be dynamic adjusted during execution based on dictionary fill ratio. When the dictionary fill ratio is still low, the window size can be expanded instead of sliding the range forward. A sliding window is commonly used in non-differential compression, and was first used by the Ziv-Lempel [ZiL77] algorithm. Obst's algorithm modified to use sliding window is similar to widely used *Deflate* algorithm [Kat96], used for zip compression format. Deflate hashes 3-grams with a 64 kibibyte dictionary and limits length of fingerprint chains using a remove-oldest eviction policy.

Instead of reducing memory requirements, Shapira and Storer [ShS03] propose the sliding window as a solution for preventing read-write conflicts of in-place reconstruction without converting copy directives into insert directives. Moving the sliding window forwards prevents referencing any position preceding the window, and eliminates any conflicting reads crossing the window boundary. However, this also prevents referencing any transposed substrings that cross the window boundary. To remedy this problem, Shapira and Storer [ShS04] propose first finding supermaximal repeats and then constructing a temporary string that contains the repeats in rearranged ordering, more suitable for the sliding window approach. Computing the new ordering requires $O((n/k)^2)$ time, where k is a chosen constant maximum length for repeats. The found repeats are sorted into a list from longest to shortest order and then numbered. The repeats are then inserted into the temporary string in order, skipping those that overlap any already inserted substring. The temporary string is then differenced and encoded with an array of the rearranged repeat identifier numbers.

3.2.4 Sparse Fingerprinting

Ajtai et al. [ABF02] describe a *checkpointing* method to improve the execution time and reduce dictionary size of the dictionary differencing algorithm by increasing the granularity of fingerprinting. Similar approach was described by Bentley and McIlroy [BeM99] for non-differential compression, to allow matching far-apart long repeats outside a sliding window. The method is based on the assumption that common substrings between reference and version strings are relatively long and occur in the same relative order. Instead of attempting to insert fingerprints for every position into the dictionary, only fingerprints calculated from positions satisfying a position selection function are inserted. Ajtai et al. suggest either fingerprinting every s’th position, or using equation $f \; mod \; s = z$, where f is a fingerprint and z is chosen by first calculating a fingerprint from random position. The granularity of fingerprinting can be chosen arbitrarily by adjusting s. We however note that latter function requires calculating fingerprints for every position and depending on distribution of f, may be triggered only in the randomly chosen position.

Sparse fingerprinting allows storing fingerprints from a range s times bigger, but the range is filled with gaps [ABF02]. Discarding some portion of fingerprints should not degrade match-finding capability by a large degree because finding any matching section of a long common substring is sufficient for extending the range of match [ABF02]. As the matching seed might be in any position within a longer matching substring, the match must be extended both backward and forward. Finding a repeat requires that at least one seed is completely contained within the repeat, implying that some repeats shorter than $s + p - 1$ symbols cannot be found (see figure 14). Bentley and McIlroy [BeM99] used sparse fingerprints as secondary differencing method with sliding window to find repeats that are outside the sliding window.

Sparse fingerprinting allows effectively choosing an arbitrary length for a string because differencing with $n/s = g$ fingerprints requires approximately the same amount of memory than differencing with all fingerprints of a string of length g [ABF02]. It becomes possible to compress arbitrary large strings by adjusting s as a function of string length [ABF02]. Choosing s can also be done dynamically based on available memory. Sparse fingerprinting can be combined with both bounded chaining and sliding window, for very long strings. The fingerprinting granularity can be denser when using flush or replace fingerprint collision policy on the dictionary [ABF02]. With keep-first policy, using too dense sampling causes the dictionary to become full and lose ability to accept new fingerprints.

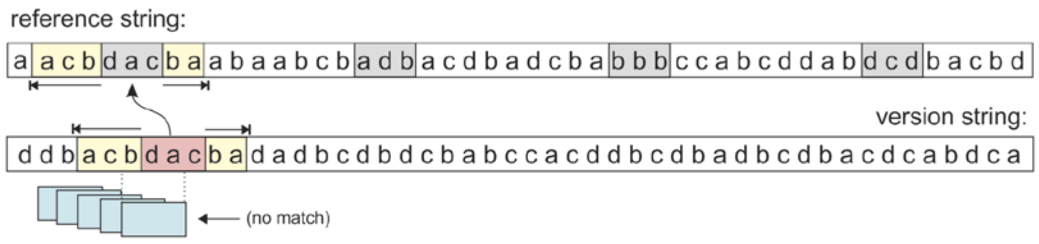

reference string:
a a c b d a c b a a b a a b c b a d b a c d b a d c b a b b b c c a b c d d a b d c d b a c b d
version string:
d d b a c b d a c b a d a d b c d b d c b a b c c a c d d b c d b a d b c d b a c d c a b d c a
(no match)

Figure 14: With sparse fingerprinting, short repeats may not be found, and repeats must be extended both forwards and backwards because fingerprints do not represent prefixes.

Ajtai et al. [ABF02] note that by iterating through the strings again an algorithm can attempt to improve the choices made on the first pass because it has more information available on the second pass than during the first pass. For example, a substring from the version string might be first encoded as added literal, but then later the search will find another substring that could be used to partially or completely encode the said substring as a reference. An algorithm might have found a matching substring that is not the longest possible match, and the longer match will be later missed because the fingerprint dictionary entry is flushed due to space restraints. We note that multiple pass approach also allows runtime analysing of several different methods, possibly on some limited initial range of the strings, and then choosing the method that appears most suitable after evaluating the preliminary differencing results. Another pass of equal time bound will not change the asymptotical time bound of the whole algorithm, but in practice it may increase the processing time experienced by a user beyond what one finds acceptable.

3.3 *Suffix Searching*

Suffix tree [Wei73] is a data structure representing and storing every suffix of a string (see figure 15). Suffix trees and are the only known method that optimally solve the problem of finding all repeats of any length p, including short repeats that dictionary search cannot find, while having $\mathcal{O}(p)$ time bound [VMG07, ABF02]. Any substring of a string is also a prefix of some suffix of the same string. A suffix tree contains all the possible suffixes, each starting from the root node. Each node of a tree represents a substring of a string. Each distinct suffix of the string has exactly one possible route from the tree's root to a leaf node. The leaf nodes represent different length suffixes. Nodes with only one child node can be progressively merged into one substring node, producing a path-compressed trie. A suffix tree can be constructed in $\mathcal{O}(n)$ time [Lar96]. Weiner's algorithm [Wei73] constructs the tree backwards, reading from right to left. Ukkonen's construction algorithm [Ukk92] creates a tree with one left-to-right pass through the string. A completed suffix tree contains at most $2n - 1$ nodes, out of which n are leaf nodes, one for each suffix [Lar96]. Because each internal node has at least two child nodes, there can be $\mathcal{O}(n - 1)$ internal nodes, each representing an intra-repeat [VMG07]. The path from root to each leaf through the internal nodes defines the complete suffix each leaf represents. Every path is different length, corresponding to the length of the suffix.

Search for any substring k of length p anywhere in the string can be solved in $\mathcal{O}(p)$ time by traversing the tree from the root, until reaching end of k [Lar96]. We describe an algorithm for finding all repeats between reference string and version string. Let T_R be a suffix tree constructed from the reference string. The version string is iterated though, starting from the beginning. For each position i, the algorithm attempts to traverse T_R from the root, using substring $[S_i \dots S_{n-1})$ as search pattern. When the first symbol from the search pattern matches the first symbol of a root node of the suffix tree, the algorithm can start progressively traversing the tree until a symbol within a node mismatches or no matching descendant node is found. The algorithm can then output the position interval between positions where traversing started and ended as a repeat for delta encoding. When there is no matching root node in the suffix tree, the symbol is not present anywhere in the reference string. Optimal differencing of any repeat length can be done in $\mathcal{O}(n + m)$ time [ShS04]. We suggest that finding intra-repeats within differences can be done by incrementally constructing a suffix tree T_D from the set of differences using Ukkonen's algorithm. After each previous substring, the next one is searched from the tree, before adding it into the tree.

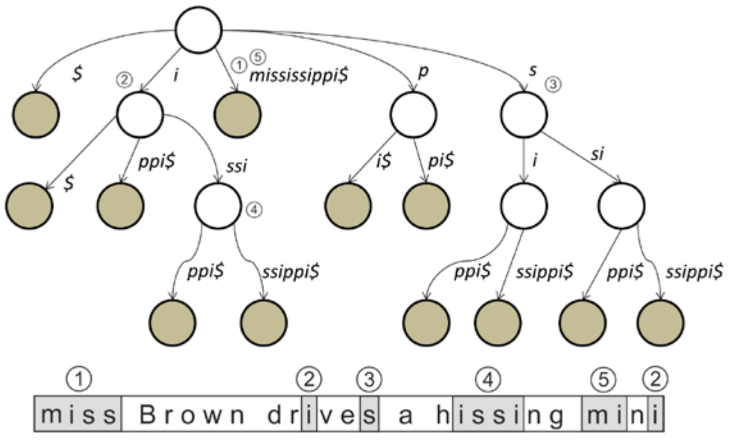


Figure 15: Finding identical substrings from two strings using a suffix tree.

It is also possible to find all supermaximal repeats directly from within one suffix tree, without iterating through the version string and specifying individual patterns to search for [Bak93, Lin07]. Baker's [Bak93] algorithm operates as follows. Duplicate prefixes of suffixes share the same nodes until where the tree branches due to a different symbol. To find a suffix of a repeat, an algorithm needs to find the deepest fork nodes (largest number of symbols traversed from the root to a node that has more than one child), $c1, c2 \in children(v), c1 \neq c2$. Finding the deepest fork node can be done in $O(n)$ time by inspecting every node in the tree and storing the longest found. Because each fork node has more than one child node, the respective substrings have different next symbols and cannot be extended forwards. The fork nodes thus represent suffixes of right-maximal repeats [Bak93]. For finding the limits of extending repeats backward, to find the positions where repeats become left-maximal, Baker [Bak93] describes an optimal algorithm. However, because maximal repeats can overlap, there can be a quadratic number of matching substring pairs. Linhard [Lin07] presented a modification to a suffix tree that allows finding the limits in $O(n)$ time. The leaves of a suffix tree are replaced with lists of *left contexts,* positions after which left-maximal repeats begin.

Suffix trees are problematic to use in practice because even though the space bound is $O(n)$, the multiplicative constant is high [Kur99]. Nodes can store starting positions of the substrings, instead of the substrings, thus requiring constant space per node [Bak93]. It is sufficient to store only one position for each internal node. However, having a linear time bound assumes having a constant size alphabet. Traversing from one node to another can be done in constant time; in this case the descendant nodes are not iterated. Constant-time lookup requires an array data structure with $|\Sigma|$ entries. The vertices from one node to another can be stored using a linked list or a balanced (red-black) tree [Bak93], but this increases the search time with a factor of $\alpha = \log_2 |\Sigma|$, increasing the time bound to $O(k\alpha)$. Space required for a node is not dependent on the number of child nodes, thus a suffix tree can be stored in $O(nt)$ bits of space, where t is the size of the node data structure [Bak93, VMG07]. Typically a well-implemented suffix tree requires more than 20 times the memory space compared to the string the tree was build from [Kur99]. Compressed suffix trees [Kur99, GrV00] allow constructing more compact suffix trees requiring ~10.1 bytes per symbol on average, but the linear space bound still prevents processing large strings without using a sliding window. Välimäki et al. [VMG07] describe a compressed suffix tree that can be built in $\mathcal{O}(n \log_2 n\, \alpha)$ time and requires $\sim 6n\alpha$ space during construction and $\sim 4n\alpha$ when completed. The creation algorithm is however 30 times slower than for an uncompressed tree [VMG07].

An improved sliding window method can be used with a suffix tree to allow differencing with constant space bound. Larsson [Lar96] describes an algorithm that uses a modified Ukkonen's tree constructing algorithm to slide the window one position at a time, instead of clearing data structures and making a longer jump forward, as is often done with dictionary searching. A suffix tree is created from an initial substring $s = [S_0 \dots S_w)$, where w is the size of the window. Moving a sliding window forward one position requires two steps (see figure 16). First, the leftmost symbol of s is removed from the tree, which is equivalent to removing the longest suffix, which is the s itself. Second, after removing the longest suffix, the tree construction algorithm is invoked to insert the symbol S_{w+1} into the tree, as if continuing to build the tree.

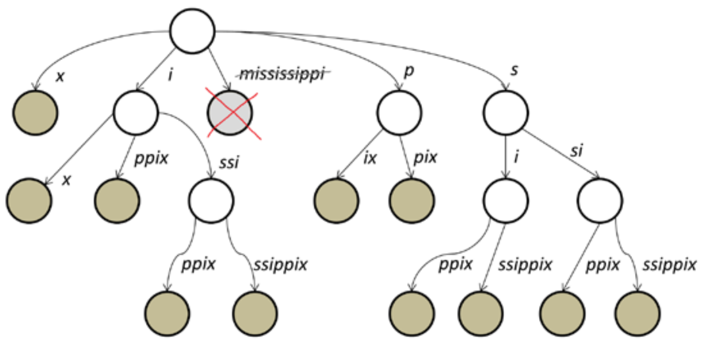


Figure 16: Sliding the window of a suffix tree one step, appending an 'x'.

Suffix array [MaM93] is a data structure semantically similar to a suffix tree (see figure 17). A suffix array contains starting positions of all possible suffixes of a string in lexicographically sorted order. Both construction of and a completed suffix array require $\mathcal{O}(n)$ space. Compressed suffix arrays require less space than the original string [GrV00], but are much slower to construct. Binary search can be used to locate the position of a given search pattern, thus a search for single pattern requires $\mathcal{O}(p \log_2 n)$ time. Suffix array can replace a suffix tree in most algorithms requiring operations suitable for differencing, by simulating operations of an optimal suffix tree with a cost of $\mathcal{O}(\log_2 n)$ asymptotic slowdown [VMG07, Lin07]. Using additional helper data structures, it is possible to eliminate the slowdown and search with $\mathcal{O}(p)$

time [AKO04]. Considering that often a suffix tree is implemented with $O(\log_2 n)$ time node traversal, and that finding the position of a repeat requires traversing to a leaf node, suffix array's time bound is equal. In our empirical test, we found suffix arrays to be faster than suffix trees in practice.

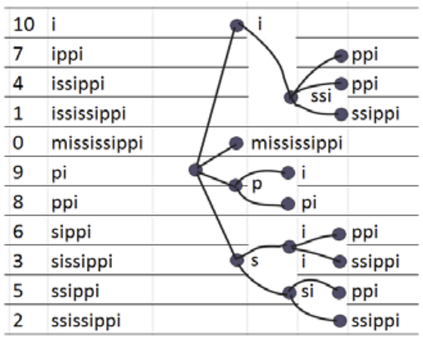


Figure 17: A suffix array on the left, with an illustrative equivalent suffix tree on the right.

The differencing algorithm for suffix trees can be used for differencing with $O(m \log_2 n)$ time bound by substituting tree traversal with binary search from suffix array. Symbol-wise comparison between search pattern and a sorted suffix is used to detect whether a suffix position is greater or less than the pattern [MaM93]. The suffix for which the binary search terminates is the longest common prefix with the pattern. Separate pre-computed tables for longest common prefixes (LCP) between successive sorted suffixes can be used to eliminate the need to restart the symbol-wise comparison from beginning of the search pattern for each comparison [MaM93]. The LCP structures can be constructed in $O(n)$ time while constructing the suffix array [MaM93]. Kurtz et al. [KAO04] describe an algorithm for finding maximal repeats from a string that can be used in place of Baker's and Linhard's algorithms.

Suffix arrays can also be used to make dictionary-based differencing faster. Agarwal, Amalapurapu and Jain [AAJ04] describe an algorithm that replaces the hash chaining used with fingerprinting with a suffix array. Agarwal et al. [AGJ05] refine the algorithm in a later paper. Because a suffix array stores suffixes having identical prefixes sequentially, an interval of a suffix array containing suffixes with identical prefixes can be thought of as a (hash) chain of colliding seed prexifes. A modified algorithm stores in its hash table an index into the suffix array and the number of hash collisions c (number of suffixes with identical prefixes). When differencing algorithm is querying the dictionary, if a fingerprint corresponds to a dictionary entry having $c = 1$, there is only one suffix of reference string matching the fingerprint, and it is verified using symbol-wise comparison. When the entry has $c > 1$, a binary search between suffix array items $A_h \dots A_{h+c}$ is used to find the suffix leading to the longest repeat. The algorithm also uses an auxiliary array and a Bloom filter to reject fingerprints that do not match, before searching from the dictionary. The other data structures are not required for the algorithm to operate, they just make it faster. Empirical testing by Agarwal, Amalapurapu and Jain indicates the array and filtering reject 97% of fingerprints before querying from dictionary.

3.4 Projective Searching

Whereas other differencing methods discussed in this thesis are designed primarily for exact pattern matching, Percival's [Per06] *projective searching* algorithm is designed for approximate differencing, even though it can be used for exact pattern matching. The idea is to find approximate matches by converting the strings into discrete signal data and then calculating cyclic correlation of the two sequences of signal data. High numeric values, *correlation spikes*, in the resulting correlative vector indicate probable positions where the two strings match: the higher the spike, the higher the probability of a match (see figure 18). The idea is that substrings either match with low edit distance, or do not match at all.

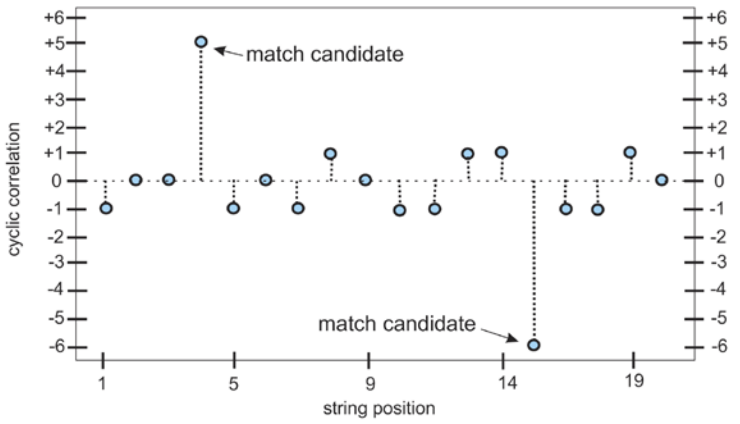
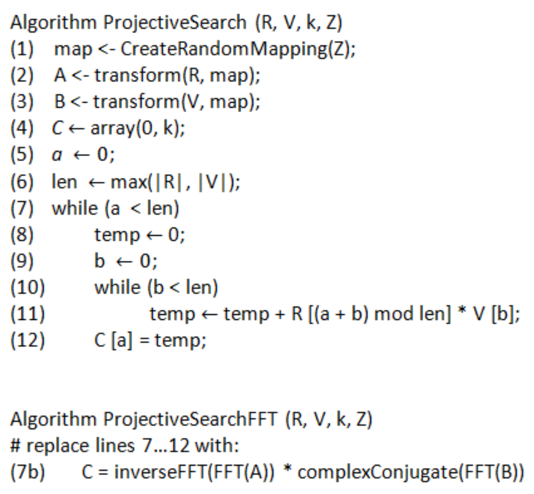


Figure 18: Cyclic correlation showing two spikes, indicating positions that are likely to contain repeats.

Percival's algoritm consists of three phases. First, both strings are converted into signal data using a randomized conversion function. The shorter string is zero-padded to make both strings equally long. Percival suggests using a function that randomly maps each symbol into either -1 or $+1$ signals, which means that the conversion is irreversible and lossy for all alphabets having $\alpha > 1$. The conversion can be done in $O(n + m)$ time. Lothar [Lot08] suggests calculating the distribution of symbols and then using the first bit of a Huffman code calculated from the distribution to decide which symbol gets -1 or $+1$ mapping, to avoid pathological cases due to randomization. In second phase of Percival's algorithm, cyclic correlation is calculated (see figure 19). First, the strings are converted into signal data according to the symbol mapping$^{(1\ldots3)}$. Then, an empty array is initialized $^{(4)}$, and for each string position, correlation is computed [5…12]. Computing the cyclic correlation requires $O(n^2)$ time and $O(n)$ space. The cyclic correlation is equivalent to computing discrete Fourier transform on the signal data [7b]. A Fast Fourier Transform (FFTW) algorithm can be used to obtain the results in $O(n\sqrt{k \log_2 k})$ time per signal table, where k is the number of projections. In third phase, a number of candidate match positions are acquired by retrieving positions of h highest spikes from the correlation vector. The h highest values represent the h most probable match positions. The matches can then be verified by symbol-wise comparison.



```
Algorithm ProjectiveSearch (R, V, k, Z)
(1)   map <- CreateRandomMapping(Z);
(2)   A <- transform(R, map);
(3)   B <- transform(V, map);
(4)   C ← array(0, k);
(5)   a ← 0;
(6)   len ← max(|R|, |V|);
(7)   while (a < len)
(8)       temp ← 0;
(9)       b ← 0;
(10)      while (b < len)
(11)          temp ← temp + R [(a + b) mod len] * V [b];
(12)      C [a] = temp;


Algorithm ProjectiveSearchFFT (R, V, k, Z)
# replace lines 7…12 with:
(7b)    C = inverseFFT(FFT(A)) * complexConjugate(FFT(B))
```

Figure 19: Pseudocode for calculating cyclic correlation [Lot08].

If the conversion into signal data is randomized, with some algorithm executions, correlative data computed using the mapping might indicate full matches at positions that do not match. Some symbol permutations also result in low signal correlation that indicates there are no matches at positions that do match, because different symbols map to the same signal value [Per06]. Therefore k should be selected large enough to avoid signal noise (random matches of symbols) from causing too many invalid results [Per06]. The larger the alphabet, the lower the probability of random match; the lossy transformation increases noise, but the reduced probability decreases noise [Per06].

Percival notes that the superlinear time required for Fourier transform can be reduced by splitting the tables into smaller sub-tables to reduce the $\log n$ component of the time bound. Solving multiple smaller tables is faster than solving the large original. The original positions are recoverable when sizes of the sub-tables are coprimes, by using the Chinese remainder theorem. Any recovered integer must be smaller than the product of coprimes used, otherwise it can be considered false result (out of range). Intuitively, the original table is shortened by adding each sub-table to the first one, using different boundary coprime for each sub-table [Lot08]. The information which coprime belongs to which sub-table is lost, but can be solved by factoring all possible positions p and ignoring those that are $p \geq n$. Using smaller primes will reduce time and space bounds, but will increase noise and probability of random spikes. Using both small and large primes lowers the probability of long repetitions of a symbol from causing similar projections. Probability of false matches can be lowered by using different randomized transformation for each sub-table, which causes false matches to occur on different positions and more likely to appear as noise (see figure 20).

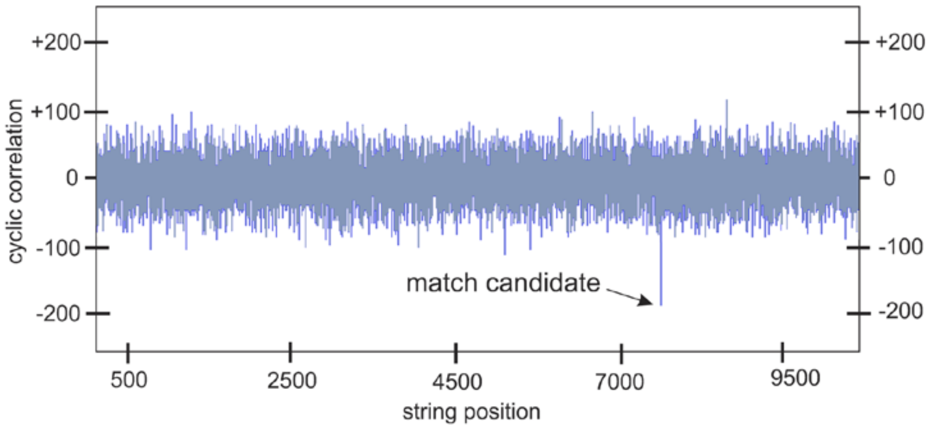


Figure 20: Random signal noise caused by computing correlation of multiple sub-tables.

Percival's approach computes cyclic correlation for non-overlapping chunks from version string, due to required computation time per search pattern. Chunking results in the alignment problem: if half of a searched chunk has changed, there is no longer a repeat for it in reference string. Correlative spike will be lower, and lowering the candidate threshold would include spurious matches. Percival thus combined using projective search with using suffix array searching [Per06]. After finding aligned chunks, the remaining intervals are searched using suffix searching algorithm. Percival notes that processors have been optimized for computing FFT, and in practice the algorithm is faster than simply looking at the asymptotic bound would indicate. The algorithm also has the trade-off between repeat matching capability and both memory consumption and computation time, making it unsuitable for long strings as the requirements grow faster than linearly [Lot08]. This was the last discussed differencing algorithm; in the next section we discuss storing the found repeats and differences into delta encoding.

3.5 Delta Encoding and Decoding

Conceptually the output of delta encoding can be expressed as an ordered list of directives representing edit operations needed to transform reference string into version string. Normally only *forward delta encoding* is generated, making it possible to reconstruct terminus string from a reference string and the delta encoding. Forward encoding does not allow reconstructing a reference string from a version string and the delta encoding (reverse reconstruction), because substrings that are no longer present in version string are not stored in the delta encoding. Bidirectional delta encoding requires storing two sets of differences [ShK10].

Delta encoding output for local differential compression needs to contain all the difference information required to uncompress V_{copy} from R and the compressed delta encoding. The lower bound for size of delta encoding is $\Omega(k \lg n)$, where k is some edit distance between the two strings, and n is the length of version string [IMS05]. A compressed delta encoding contains zero or more coding tables for secondary compression and all the encoding directives compactly encoded. It can also contain a digest checksum for verifying the identity of reference string and another checksum for verifying that uncompressing V_{copy} produced an identical copy of original V. The decompressor calculates and verifies the checksum of V from the completed terminus string.

Tichy notes [Tic84] that a change edit operation can be simulated by a combination of deletion and insertion and that any such combination can be further substituted with a combination of deletion and copy, if corresponding content is available in the reference string. Deletion can be simulated by simply omitting the substring from delta encoding. Thus, two kind of explicit encoding directives are needed [Tic84, BuL97]. A *copy* directive can be defined as a constant-cost special insertion operation from an external source. An *insert directive* instructs to add a literal string, and is essentially a special copy directive that has an implicit source position within the delta encoding itself. Deletion operation is implicit: the range of a removed substring is simply omitted from the copy directives, and nothing is written when reconstructing. We note that it would also be possible to use explicit add and remove directives, instead of the copy and add, thus having an implicit copy directive. When approximate (sparse) repeats are encoded as references, a third explicit directive type of *repair* is needed as a special version of add to fill in the errors within in the repeat [Per03].

For copy directives, source position and length of copy is stored. Insert directives store the length of copy and the substring to be copied. If the added string is stored adjacent to the directive itself or separately in identical sequence to the directives, explicit source position does not need to be stored, as a substring always follows the preceding one. In case the directives have been reordered for in-place reconstruction, destination position is also stored. Intra-repeats from within a version string can be expressed using copy directives having source positions outside the range of reference string. If the reference and version strings are identical, a single copy directive is sufficient to encode the whole version string. If the strings are completely different, a single insert directive may store the complete version string.

Ajtai et al. [ABF02] describe a conceptual encoding using add and copy directives, as follows (see figure 20). An insert directive is encoded as a sequence of three elements: a command identifier, length of a substring, and the substring. A copy directive is similarly encoded as a sequence of three elements: a command identifier, a source position, and length of the substring. For calculating a cost for directives, an insert directive's cost is the length of the substring and copy directive's cost is always one. The cost of delta encoding is the sum of the cost of all directives included in the encoding. The following transformations do not change the total cost: 1) an insert directive adding a substring is substituted with a sequence of insert directives, each adding one symbol from the substring; 2) an insert directive adding a single symbol substring is replaced with a copy directive to copy the symbol from reference string. This simple cost measure implies that when a reference string contains at least one of each symbol present in version string, the delta encoding could be done with just copy directives and the minimum cost delta encoding would be one with minimum number of directives.

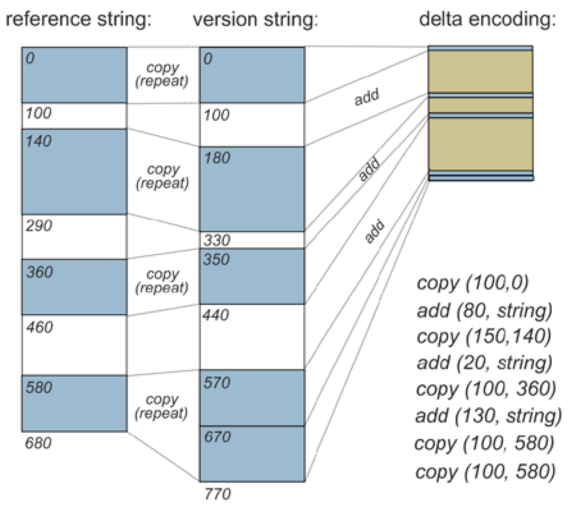


Figure 21: Conceptual delta encoding.

The encoding itself requires additional space for information such as which type directives are, what the source position in the referenced string is, and how long the referred substring is. This extra information always requires non-zero amount of space. In practice, the cost function of directives is thus more complex. We first discuss storing uncompressed directives using constant-size integers for positions. In this case, copy directives itself require constant space and are equally good choices for encoding, regardless of source position or content [ABF02]. Insert directives require constant space, plus space proportional to the length of inserted substring [Obs87], when substrings are stored without compressing them. Encoding fewer but longer references allow for better compression ratio than encoding the same substring partitioned into greater number of shorter references [ABF02].

The constant cost for a copy directive is greater than cost of extending an insert directive by one symbol, in practice by a factor of 8…16. Unencoded absolute position or length information requires $\Omega(\lceil\log_2 n\rceil)$ bits. When a differencing algorithm is capable of finding short repeats, encoding a refence to a repeat could produce longer encoded output than encoding the repeat as an extension to neighboring insert directive [ABF02]. For example, a sequence of (add, reference, add) could be more efficiently encoded as a single insert directive, if the copy directive requires more space than the string it refers to. The directive space requirement forces to ignore short repeats and allows some inefficiency in finding the repeats [ABF02]. Due to this, e.g., dictionary searching is able to use longer seed prefixes [ABF02]. A simple encoding scheme using constant-size integers may require discarding relatively long repeats, up to 18 symbols [Obs87, Chr91]. What is too short a repeat to encode depends on the compression efficiency of the encoding relative to the original string encoding [ABF02]. Substrings of one symbol are always too short to encode as references, otherwise it would be possible to compress random string content using references. Most symbols would have a matching single-symbol substring somewhere.

Absolute position encoding cannot consume fewer than $\lceil\log_2 n\rceil$ bits, but when there are multiple directives, they can contain positions relative to neighboring directives [KoV02]. Copy directives with relative offsets thus require $\lceil\log_2 p + \log_2 l + d\rceil$ bits, where p is relative position, l is relative length of the copy and d is size of directive type information [KoV02]. Copy directive cost is thus dependent on both the source position and length. Two references to the same substring may have different cost depending on from where and when it is referred to [TMS02]. The closer a position is to some prior processed reference, the smaller the cost to encode the relative offset [TMS02]. Additionally, secondary compression can be used for substrings for insert directives. This produces varying encoded lengths for substrings depending on current probability prediction context, thus causing directive sizes to become content-dependent. The compression context may contain information outside the substring, if the information is also available in reference string. Compressing the substring of an insert directive makes the directive’s length only indirectly dependent on the substring length [ShS03]. In practice, cost of an insert directive can be known only after compressing it. Choosing whether to encode a reference or an insert directive cannot be based on repeat’s length prior to encoding, thus a differencing algorithm cannot know in advance how short repeats it could ignore [ABF02]. Because directives require different amount of space depending on relative positions and content, the cost functions are non-trivial. Garey and Johnson [GaJ79] have proven that choosing the optimal set of encoding directives with varying cost measure is a NP-complete problem [KoV02, ShS03].

When strings are locally and randomly accessible, an algorithm is able to read the same subsection of a string again and again. This allows an algorithm to keep only partial information in memory, because the algorithm can later re-read a subsection of a string if the algorithm needs to know the full content again [ABF02]. A differencing algorithm can thus output only the string positions instead of copies of substrings, and an encoding algorithm will get as input parameter a list of value triplets $\{ type, position, length \}$ representing repeats and differences. Delta encoding can be performed by processing the input triplets in order. For each input, either an add or a copy directive is output with non-zero length. For insert directives, the substring is copied into the delta encoding. Detailed description of secondary compression of the output is outside the scope of this thesis. Agarwal et al [AGJ05] note that encoding directive metadata separately from substrings requires less space because each data type can have their own prediction context for secondary compression. However, this results in multiple output streams, which requires temporary space for the additional streams, as their final length is unknown prior to encoding. Trendafilov, Memon and Suel [TMS02] have modified an existing deflate compression algorithm implementation for delta encoding. They use two position reference variables instead of one (one for each string), and encode the substrings separately from copy length tables using two Huffman tables.

Decoding the delta encoding cannot be completed with a single reading pass through both reference string and terminus string. Copy directives may refer to any position of reference string in any order, due to transpositions [Tic84]. Reordering of directives can reduce the time required for reconstructing the terminus string due to reduced amount of random access [BSL02]. Reordering however requires explicit destination positions for directives, which require additional space. Both Tichy's and Obst's greedy algorithms could be optimized for decoding by searching the search for longest repeat from the position the most recent repeat was previously encoded, then wrap-around to the beginning of string when reaching the end, and stop searching when reaching the starting position again [Tic84].

We describe a standardized encoding format for delta output as an example. The VCDIFF generic differencing and compression data format [KoV02, KMM02] specifies a simplified encoding specification suitable for local differential compression. The specification focuses on differencing two local files and does not attempt to generalize enough to be suitable for remote differential compression; it does not define concepts compatible with fingerprint set reconciliation, such as recursive fingerprint comparison using multiple communication rounds. The emphasis is on simplicity instead of efficiency: header structures are constant-length, one header for each file and one for each encoding window. The specification does not specify any secondary compression scheme for symbols. Instead the specification recommends using an external compression algorithm to compress the output after completion. There are no restrictions on differencing algorithm used but the specification assumes that dictionary searching with non-overlapping sliding windows is used. The specification assumes that reference and version string are catenated into one superstring, and uses positions greater than length of reference string to refer to version string. A decoder does not need to be aware of any sliding window heuristics used for encoding [KoV02].

The format uses base-128 encoding for numeric digits. The most significant bit is reserved as an indicator whether the number contains more bytes. The encoding thus allows using 1…n bytes per number. In addition to the required add and copy directives, a “run” directive is specified as repetitive add. All directives have two parameters. An insert directive has length of added substring and the substring itself. A copy directive has position of substring within string and the length of the substring. The reference can refer to either reference string of version string and it is differentiated by the position (position exceeding reference string length refers to version string, as if the strings were catenated). The encoding allows optimizing repeating references by specifying lengths that exceed available string length. A run directive has count and a byte to be repeatedly inserted. Uncompressing can be done one directive at a time from left to right, by simply either adding a substring or by copying a referred substring. Encoding and decoding are both possible in $\Theta(n)$ time [KoV02].

3.6 Tolerating Superfluous Differences

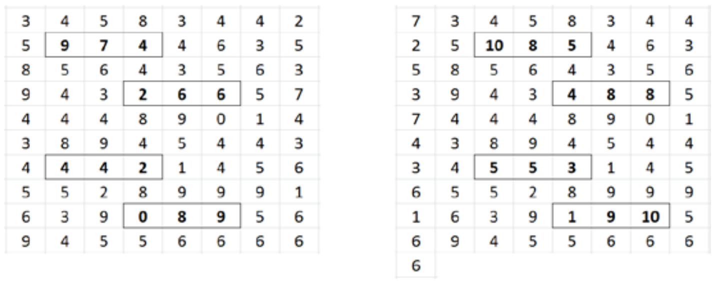
A minor change on the application-level can cause significant string-wide changes on the symbol-level, which prevents differencing from finding repeats efficiently. Percival [Per06] has divided the types of changes into three classes. Zeroth-order changes are semantically non-modifications and caused by issues such as inserting timestamps or counter values into a string when saving it into a file. First-order changes are the actual changes caused by editing. Second-order changes are changes indirectly caused by first-order changes, such as changed relative offsets that often propagade through the string. For example, recompiling a binary application with minor source code changes often produces a machine-language output that contains a large number of small differences spread around the binary output. Single inserted machine instruction pushes all following code forward and compiler must change all affected address values (see figure 22), thus dividing long repeats into short fragments [Per06, MGC07]. Minor code changes can also cause cascading changes into registry allocation instructions. Modifications to inlined[4] functions replicate the changes into each copy of the function code and cause progressive alignment changes. Adding a single line of new code can change up to 10% of all symbols in an executable [Per03]. As number of second-order changes becomes larger, matching longer repeats becomes increasingly improbable.

Application-level domain knowledge is required for these cases [Per06]. Some universal fully-reversible preprocessing methods can be used to increase apparent redundancy by transforming and restructuring the substrings to be more similar to each other, thus transforming the content into a form that can be more easily compressed [Ski06]. One such method is the Burrows-

[4] Inlined function is such where compiler generates separate copies of binary code for function calls, instead of generating jump instructions to shared code.

Wheeler transform (BWT) [BuW94], which rearranges the symbols of a string in $O(n)$ time and space. The transform does not change the length of the string. Tridgell [Tri99] applied BWT to text documents having different line-separator symbol sequences (e.g., ‘\r’ in Unix versus ‘\r\n’ in Windows) and noted a speed-up factor of 27 for synchronization.

Knowledge on executable binary format can be used to completely disassemble binary code into source code format and deterministically encode the changed offsets, or a modified compiler can be used to generate list relative address offset location [Per06]. Google’s *Courgette* algorithm [Ste09, Ada09] disassembles the executable into a simple proprietary format. All jump and address instructions are analyzed and replaced with symbolic references. Other binary machine code itself is not analyzed and left untouched. Both updated executable (version string) and the existing executable (reference string) are processed, which results in similar strings, because the instructions are similar, only the address values had changed. These pre-processed strings are then compressed differentially using publicly available software (bdiff). Resulting delta encoding is an order of magnitude more compact than without preprocessing [Ada09]. Updated executable is reconstructed on users system by first disassembling the user’s existing version and then using an assembler.



3	4	5	8	3	4	4	2
5	**9**	**7**	**4**	4	6	3	5
8	5	6	4	3	5	6	3
9	4	3	**2**	**6**	**6**	5	7
4	4	4	8	9	0	1	4
3	8	9	4	5	4	4	3
4	**4**	**4**	**2**	1	4	5	6
5	5	2	8	9	9	9	1
6	3	9	**0**	**8**	**9**	5	6
9	4	5	5	6	6	6	6

7	3	4	5	8	3	4	4
2	5	**10**	**8**	**5**	4	6	3
5	8	5	6	4	3	5	6
3	9	4	3	**4**	**8**	**8**	5
7	4	4	4	8	9	0	1
4	3	8	9	4	5	4	4
3	4	**5**	**5**	**3**	1	4	5
6	5	5	2	8	9	9	9
1	6	3	9	**1**	**9**	**10**	5
6	9	4	5	5	6	6	6
6							

Figure 22: Second-order changes caused by changed offsets.

Another approach is to find approximate repeats and then include additional information for repairing the non-identical symbols included in the approximat repeats. Percival [Per03] implemented a compressor that uses dictionary searching for finding initial short seed prefixes, but when attempting to expand the found repeats, ignores mismatches up to 50% of number of symbols in the extented range. Only repeats that are a minimum of 8 symbols longer than the longest previously found repeat are accepted as new best choices. To prevent including heading or trailing non-matching sequences, expanding is stopped and non-matching symbols excluded when a contiguous sequence of k non-matching symbols is detected [MGC07]. During construction of the delta encoding, the mismatching substrings within approximate repeats are sequentially inserted into a separate section of the delta encoding. For each mismatch, copy directive is split and an insert directive is encoded with a length.

The exediff algorithm [BMM99] uses the same method but improves compression by predicting symbol values for second-order changes, based on knowledge of positions of machine-code jump opcodes. When a relative offset has changed value, it is likely that close-by opcode offset values changes have changed by similar amount [MGC07]. Exediff calls differencing pre-matching, as the exediff algorithm iteratively attempts to decode sections during a *value recovery* phase to test the predictions and compares the length of encoded copy directive output against using an insert directive [MGC07]. Approximate matches that require more bits to encode are then ignored and encoded as insert directives. Approximate matching may find spurious matches (that match by random change), some of which can still be encoded using references [MGC07].

Motta, Gustafson and Chen [MGC07] make an assumption that second-order mismatches have regular patterns and can be predicted, without content-dependent knowledge. They describe an algorithm that attempts to predict the changes based on k previous mismatches, stored as cached information. Delta decoder needs to recreate the same prediction data structure. They suggest using a cost function to estimate the relative ranking of approximate repeats, because length of an approximate repeat does not alone describe the goodness of a repeat. One simple cost function is $c(x) = b / r$, where b is number of bits needed to encode the directive and r is the length of the approximate repeat. The algorithm attempts to match a mismatch against earlier mismatch in the cache. Multiple similar mismatches can be encoded more efficiently as the same repair operation can be repeated.

Percival [Per06] later improved his algorithm in bsdiff implementation. The algorithm uses projective searching for differencing and allows some number of second-order changes for matching repeats. Error correction codes are used in delta encoding to allow recovering from these changes during decoding. Instead of attempting to exactly match the repeats, Percival's method treats small number of changes as transmit errors that can be repaired. The algorithm shifts the relative position of version string, and approximate repeats that have relatively close positions in the strings are considered to contain second-order changes. Delta encoding with error correction codes allows creating a single patch file that can be used to synchronize against multiple slightly differing reference strings [Per06]. Such cases are common in the open source community where different versions of a compiler are used to compile applications, resulting in slightly different binaries for the same application source code [Per06]. Unfortunately, neither Percival's algorithm's implementation nor its encoding format has been published.

Both Percival's bsdiff and Motta, Gustafson and Chen's algorithms produce compression ratios that differ by less than factor of 1.2 from results of platform-specific implementations [MGC07]. Even though we used executable binaries as an example, the approximate differencing methods are generally suitable for strings having second-order changes with repeating similar modifications. The algorithms are not suitable for cases such as when a binary document is converted from little-endian to big-endian integer encoding.

4 Remote Differential Compression

This chapter discusses known methods for remote differencing. Common case is that two strings needing to be synchronized are stored on two separate hosts. Neither host knows anything about the string stored on another host. One host cannot read both strings and perform differencing on its own; instead two separate instances must exchange messages, transmitting little as possible data between the hosts. Temporal cost of accessing a string on another host is high due to to access latencies such as network round-trip time or security layer handshake negotiation. Attempting to read multiple pieces of a string from other host may require more time than simply transmitting the whole string compressed using non-differential compression. An algorithm should infer which sections of the strings are different and synchronize the strings without transmitting much more data or using more time than copying a non-differentially compressed version string would require.

In seection 1 we discuss solutions to the general problem of inferring which sections of strings are different, without being able to read the actual strings. Common feature of the solutions is computing some summary value (fingerprint) of a substring, and then using the summaries to propabilistically infer which intervals of the strings are different. In section 2 we discuss fingerprint collision probability and choosing a hash function for fingerprinting. In section 3 we discuss *chunking*, where fingerprints are used to represent arbitrary long chunks (substrings), thus requiring fewer bits for communication but enforcing a long atomic unit of change. In section 4 we discuss different methods for avoiding chunk alignment problems. Solutions include using variable-length chunks whose division is defined by content, and comparing all possible fingerprints of a string against the set of fingerprints from other host. In section 5 we discuss algorithms for differencing the ordered multi-sets of fingerprints, using interprocess communication. In section 6 we discuss the synchronization algorithm, which includes producing the delta encoding and reconstructing the terminus string. Finally, in section 7 we discuss improvements to the fingerprinting scheme to allow more fine-grained chunking and using fewer bits per fingerprint.

4.1 Probabilistic Method

A synchronization algorithm must minimize communication between hosts. Non-zero amount of data needs to be communicated between the two hosts to infer which parts of the strings are different. Communicating less information than the strings contain implies that differencing needs to be probabilistic: remote differential compression cannot achieve full certaincy of succeeding [Tri99]. Computer hardware in general also has a given probability of hardware error, depending on system, so probabilistic operation is not necessary a problem. For discussion both for and against the probabilistic approach, see papers by Black [Bla06] and

Henson [Hen03]. There is no deterministic solution requiring fewer bits of communication than copying a conventionally compressed version string to the host having the reference string, and differencing locally [Yao79].

The hosts exchange metadata about the strings and then infer which substrings need to be transferred to the other host. In the following discussion, the host storing the reference string is denoted reference host, and the host storing the version string is denoted version host. Hosts' roles as client or server in user's perspective do not matter. Worst case is that the strings are completely different, requiring that the version string be copied completely. Thus, worst case communication bound is $\Omega(g + h)$ bits, where g is the size of conventionally compressed version string, and h is number of overhead bits required for remote differencing. Orlitsky [Orl91] gives h a lower bound of $\Omega(\lceil \log n \rceil)$ with a single communication round and $\Omega(\lceil \log \log n \rceil + 1)$ bits with two communication rounds. Single-round synchronization can require exponentially more data than multi-round synchronization, but only twice as much if the edit distance is known in advance [Orl90]. Two messages is close to the optimal [Orl93], but three messages is still not optimal [ZhX94]. An algorithm is communication efficient if it transmits at most $k \lg^c(n)$ bits, where c is some constant [IMS05]. This efficiency measure ignores number of communication rounds and common network protocol overhead, such as TCP, IP and TLS headers. For measurement purposes, string length can be converted to time (or vice versa) by multiplying with a constant representing some chosen transmit bandwidth, to allow simpler comparison with a single measure.

Remote differential compression is not required if either of the hosts has both the reference string and version string available. Also, it is possible to avoid using remote synchronization when only a known finite set of possible different reference strings exist. For example, a server offering software update services can store all known versions of the binaries along with digest checksum (hash) calculated from each reference string. A client can then calculate a corresponding digest from its reference file and transmit only the digest to the server. The server can then match the digest against all known ones and transmit a pre-compressed local delta encoding to the client.

Early remote differencing algorithms were intended for detecting and correcting errors in remotely replicated strings. Purposeful modifications can be viewed as as random errors. Orlitsky [OrV02] noted that problem of using error correction codes for repairing transmission errors and the problem of synchronizing two remote hosts with similar data are closely related. The reference string can be seen as a corrupted copy of the version string, and error correction is used to repair reference string into version string [STA03]. Minsky, Trachtenberg and Zippel [MTZ03] note that correcting errors in a string is more difficult than in a set of integers because the ordering of symbols is important and the length of a string is unbounded.

Metzner [Met83] described an algorithm for locating errors between two remote strings by constructing a tree structure of parity values calculated from chunks. The algorithm divides the two strings into equal-length chunks and then calculates a parity value for each chunk, separately and independently on each host. From these parity values, disjoint sequences are formed and another level of parity values is calculated from each sequence, forming the tree of parity values. Metzner's implementation used only two chunk parity values to calculate each higher level parity, thus creating a binary parity tree. The parity scheme is modified for each level to avoid two changed parity values from cancelling each other out. String positions are used as seeds for a pseudo-random number generator for the parity functions. Binary search and recursive transmitting is used to compare the chunk parities. The algorithm requires transmitting $(\log_2(n/c) + 1)(b + 2) + c$ bits using $\lceil \log_2(n/c) + 1 \rceil$ communication rounds for locating and correcting a single differing chunk on either branch from the root, or twice the amount if there is a difference on both binary branches (where c is the size of chunks, and b is the number of bits per parity). Metzner later [Met91] improved the efficiency of the algorithm by probabilistically transmitting additional parity values (from next levels) on each round, thus requiring less communication rounds if the additional parity values solved the error location problem. However, because the algorithm was intended for detecting errors, it did not consider substring insertions and removals and the resulting alignment problem. Due to constant-length chunks, alignment changes lead to large number of differing parity values.

Schwarz, Bowdidge and Burkhard [SBB90] improved Metzner's algorithm by adding capability of identifying single missing or added chunks. The algorithm uses supersignatures, calculated from a contiguous sequence of parity values. Finding single missing pages is attempted by shifting the parity sequence interval from which the supersignature is calculated, up to d steps to either direction. Each time the algorithm proceeds to the next level down the parity tree, d is decreased by one, because there must be something differing since the parities did not match. Choosing d sets an upper bound for detectable differences. Their algorithm cannot match transposed chunks, unless the chunks are within d distance from each other.

Differencing the sets of parity values can also be modeled as special case of non-adaptive group testing problem, where d items are different. Madej [Mad89] describes an algorithm for differencing with arbitrary d. The algorithm requires $\lceil ((2 \log_2 n)/a)^a \rceil$ additional composite parity signatures, where $a = \lceil \log_2(d + 1) \rceil$. The signatures are calculated using XOR binary logic, leading to a problem that multiple changed bits can mask differences. Space bound of the algorithm is $O(e(d + 1)^2 \log c)$, where c is the number of chunks. Lower bound for probability of success is $1 - 1/d!$. Again, transposed chunks cannot be matched.

We exclude further discussion on these parity-based methods because the methods are unable to find transposed repeats, similar to local differencing by finding a longest common subsequence. Later algorithms are based on the fingerprinting scheme used for seed prefixes with local differential compression, which can be used to identity longer substrings by increasing fingerprint strength. Thus, a relatively short bit string allows representing arbitrary large amount of data and enables detecting substring inequality in constant time [Tri99]. Computation of each fingerprint is independent of others [Tri99]. Changes to multiple chunks do not cause masking fingerprint bits nor prevent detecting the differences. The idea of transmitting hashes to remote host was mentioned in Heckel's paper [Hec78], including using recursion to initially difference using longer atomic units of change. By using fingerprints to represent chunks, the problem of remotely differencing two strings becomes a problem of differencing the two ordered multi-sets of fingerprints.

Verifying by symbol-wise comparison after matching two fingerprints is not possible due to transmission cost between hosts. Thus, each fingerprint calculated from unique chunks must be unique among all fingerprints calculated from the reference and version string [Tri99]. More complex hash algorithms must be used to calculate more uniform distribution of hashes and thus lessen the probability of a hash collision. However, since much fewer bits are used for a fingerprint than the string it identifies, it is impossible to guarantee that the hash function produces a unique value for every different string [Tri99, You06]. More complex algorithms are computationally more demanding, which means that selecting a hash function is a compromise between reliability (collision improbability) and performance.

Algorithms designed for remote synchronization are often not practical for local differential compression because they require more execution time and compression ratio is comparably bad due to necessity to use longer atomic unit of change. Remote synchronization always requires transmitting data to detect properties of the strings, whereas local differential compression does not. Still, chunking can be used and has been implemented for local differencing, because it allows differencing arbitrary long strings by increasing the chunk length as a function of string length [BeM99]. Chunking can be viewed as sparse fingerprinting with long seed prefixes. Chunk fingerprints can be stored for later use to avoid reading a string; for example, backup software could use the stored fingerprints to difference against a reference string that itself is no longer accessible as it has been moved to safe location.

4.2 Fingerprint Collisions

Fingerprint collision can cause *silent corruption* [Hen03], where the terminus string is not identical to version string, and the error is not detected. The rate of failure is dependent on fingerprint size, number of chunks, and mathematical qualities of the fingerprinting function [Bla06, Tri99]. In universal hashing the purpose of hashing is to distribute values among given range to speed up processing and the number of distinct objects can be relatively large compared to number of possible hash values h [You06]. Universal hashing assumes hash collisions are common and can be resolved with hash chaining. Fingerprinting without possibility to verify requires that all fingerprints are unique, meaning that collisions are not allowed at all. The number of distinct objects must be much smaller than the range of possible fingerprint values ($n \ll h$) [You06]. This implies that required fingerprint size is dependent on strings' length and chosen chunk size [Tri99]. The longer the string, the more chunks there will be, when the average chunk size is assumed constant. The more chunks there are, the higher the probability of hash collision. The probability of synchronization failure is approximately cp, where c is the number of unique fingerprint comparisons and p is the probability of a collision [Tri99].

When a collision occurs, synchronization algorithm will wrongly assume the chunks are identical and that it already has a copy of the chunk, even though it does not. Synchronization will produce a corrupted version string that has some chunk substituted with another different one, both having the same fingerprint. Thus, unlike with local differential compression, a weak rolling hash function cannot be used (by itself) because comparing the chunks later with symbol-wise for verification is not possible. The fingerprint function has to produce checksums that have a probability of collision less than probability of other unavoidable failures, such as unrecovable hardware error [You06]. For example, if the fingerprinting function simply interpretes the next four symbols as the fingerprint of a chunk, a hash collision would occur each time the same four symbols would occur in the beginning of a chunk.

An ideal fingerprinting function's output can be considered random and it is expected to produce fully uniform distribution of output values among the output value range [Bla06]. Probability of a hash collision, one random block having the same fingerprint as another, is approximately 2^{-b} when using b bits for the fingerprint [SNT04]. About $\log_2 n$ bits is needed for equal change of collision or non-collision, and each additional bit decreases the probability of collision. The probability of two fingerprints being the same is approximately $1 - e^{\frac{-k(k-1)}{2n}}$ [SNT04, Bla06], where k is the number of fingerprints. Actual real-life hash functions' output cannot be considered random and the probability of collision is higher because hash function implementations are not truly ideal [Hen03]. Hash collisions are deterministic and when a strong fingerprinting function causes a collision on different chunks, it will prevent from synchronizing the strings [Hen03]. Retrying does not help, as the fingerprint output stays the same. Evaluating hash collision properties on arbitrary content is non-trivial [Tri99]. For example, a strong hash function H modified with an exception of $H(0) = H(1)$ has almost the same collision probability as the non-modified function [Hen03]. A random polynomial functions used for Rabin fingerprinting in Tridgell's rsync was found to have 5 bits of strength less than ideal [Tri99]. Improved polynomial hash functions produced 3 bits less than ideal results for 32 bit hashes. To compensate for the limited randomness, some chosen number b of additional bits is needed for the fingerprints. A total of $\log_2((n+m)/c) + b$ bits is needed per fingerprint when comparing chunks having an average length of c symbols [IMS05]. Irmak, Mihaylov and Suel [IMS05] suggest using $b = 10$ additional bits.

Because designing a strong hash function is difficult, cryptographic hash functions are typically used for chunk fingerprinting [Tri99, You06]. They have been designed such that any kind of data modification has very high probability of causing a change in the calculated hash value [You06]. They have been designed to be resistant to malicious tampering, which means it is difficult to produce hash collisions even on purpose [You06]. Instead of security properties, other reasons for using cryptographic hash functions are consistency and availability [Tri99]. Popular cryptographic hash functions also have performance advantages because there are often highly optimized implementations available [You06]. There are specific processor instructions added into hardware to accelerate calculating popular hash functions, such as AES instruction set for x86-64 family of processors [Int10]. Different implementations of a given hash function can have a factor of 10 difference on total execution time [You06].

Popular cryptographic hash functions such as SHA-1 and SHA-512 [NIS08] are being studied extensively. When vulnerabilities (ways to produce hash collisions) are found, improved hash functions become recommended instead. Currently recommended hash functions have gone through intense cryptanalysis and are believed to have strong collision resistance. Application implementations will benefit from upgrading the fingerprinting functions to the new stronger cryptographic hash functions when available [You06]. Cryptographic hash functions are resistant to preimage attacks, where content is fabricated to produce a colliding hash value [Bla06]. Invertible, non-cryptographic hash functions allow fabricating content to produce a given hash value relatively easily. Tampering resistance is not required for compression purposes because purposeful tampering is not usually an issue [Tri99]. However, for cloud storage where data deduplication (a form of differential compression) is used between accounts, a malicious user could cause hash collisions that cross user account boundaries [Hen03]. For example, SHA-1 collisions can be found in 2^{69} function calls for 160-bit digests, whereas the expected number of calls (cryptographic strength) is 2^{80} [PCT11].

When synchronizing long strings, there can be millions of chunks and even small changes in fingerprint size overhead will be meaningful in practice. Compression ratio can be improved by reducing the strength of the checksums dynamically as a function of string length, to optimize the amount of data needed to be transmitted [IMS05]. Checksum strength can be reduced by reducing the hash values to a finite field of $[0 \ldots modulus)$ using modulo arithmetic, or by simply dropping off a number of bits. We note that unless either checksum strength or average chunk size is dynamically increased, synchronization of long strings becomes improbable to succeed or finally impossible as the number of chunks becomes larger than the number of possible different fingerprints.

To detect corruption due to hash collision, Tridgell [Tri99] suggests calculating a separate digest checksum from the entire original version string. The reference host can then calculate a corresponding digest from the reconstructed version string and compare the digests for verification. Because a strong digest checksum is relatively small compared to total amount of data transmitted, a much stronger checksum can be used than for chunk fingerprints. A digest checksum increases the probability of detecting corrupted reconstructed version string from zero to $1 - 2^{-b}$ if the checksum functions are assumed ideal [Tri99]. Detecting collisions makes it possible to weaken the chunk checksum strength used for fingerprinting. Reducing the chunk fingerprint size reduces the overhead of the protocol, but may require additional differencing attempts with stronger fingerprints if a collision is detected. Without the digest, the chunk fingerprints need to be strong enough to never cause collisions with very high probability. With the additional digest, the fingerprints need only be strong enough to rarely cause collisions, if it is assumed that the digest allows detecting failure with high probability. We note that a separate checksum also protects against network transmit errors (noisy transmit channel) and hardware errors. Network data transfer using TCP protocol is prone to errors because simple 16-bit cyclic redundancy check is used for detecting transmission errors. The acceptable collision frequency is implementation specific (e.g., how frequently retries are allowed).

Lacking digest strength combined with weak chunk fingerprints may cause a failure when synchronizing certain content. For example, it is possible to fabricate strings that old version of popular Rsync tool cannot synchronize due to using MD4 [Riv90] for the strong fingerprint [Per06]. Version 3.0 thus changed to using MD5 [Riv92] instead of MD4 [Tri08]. When the digest verification fails, some failover semantics are needed. One solution is to perform the synchronization again using a stronger fingerprint size or different fingerprinting function [Tri99]. Current cryptographic hash functions do not allow reparametrization like the simpler functions, such as Rabin fingerprinting based on random polynomial [Tri99]. Another fallback solution is to treat all chunks as different and re-transmit the complete version string compressed using non-differential compression [Tri99]. Fingerprinting, even with very strong digest checksums, can never provide 100% probability of either successful reconstruction or detection of corruption.

4.3 Chunking

We describe a generalized method of differencing for basis of discussion, based on algorithms by Tridgell and McKerras [TrM96] and Teodosiu et al. [TBG06]. For inferring which substrings are different between reference and version strings, the strings are deterministically divided into non-overlapping chunks (short substrings) of some length using identical heuristics on both hosts [BBG06]. *Division points* are positions of a string specifying a starting position for a chunk. When no division points can be found, the whole string is the only chunk. Both hosts may operate independently, and it is assumed that performing the chunking the exact same way on two similar strings will produce similar chunk cutpoints [BBG06]. The simplest division heuristics is to use chunks of fixed length c. For each chunk, a fingerprint is then calculated using some strong hash function. Each fingerprint uniquely represents the corresponding chunk, but only in the context of the two strings. These checksums requiring much less space than the chunks themselves are then be transmitted to the other host, along with parameters of the chunking method. The other host can then calculate the respective fingerprints from its own local string and compare them against the set received from another host. Hosts assume that identical fingerprint equals identical chunk. Only one communication round is required between the reference and version host [TrM96]. With this basic algorithm, synchronization requires sending at least $\Omega(n/c)$ fingerprints (where c is the average chunk length) from reference host to version host, even when the strings are identical [Tri99]. Chunk fingerprints can be saved and later used for differencing without having the reference string present any more. This method can be used for incremental backups, where fingerprints calculated from current string are compared against fingerprints saved during the last backup run.

Chunking has an intrinsic problem that if one single symbol of a chunk is modified, the whole chunk is considered different [Tri99]. Because hash function's output is expected to be pseudo-random for differing inputs, changing a single symbol is expected change the fingerprint completely, making approximate matching impossible. Subsequently, if every chunk has a single symbol modified, then the whole string will be considered different [SNT04]. Due to longer atomic unit of change, chunking is not suitable for strings with second-order changes, because the changes may occur with dense granularity. Remote differential compression of binary executables often fails to reduce the transmit cost over conventional compression, because systematic second-order changes have changed most chunks. Decreasing atomic unit of change (average chunk size) would increase protocol overhead from fingerprints, exceeding the amount of space saved by compression [Tri99].

Another issue with chunk-based differencing is the relative alignment of the repeats between the two strings [Tri99]. If the strings are divided into fixed-size chunks and a single symbol is added to or removed from either string, then none of the chunks that contain one or more symbols from the shifted range will match anymore, if the chunks are compared trivially in the same sequence [You06]. Similarly, all chunks that contain symbols from transposed ranges will have different checksums when compared against chunks in the same relative positions. We note the alignment problem is analoguous to the alignment problem with naive symbol-wise comparison with local differential compression, if symbols from chunks are viewed as bits and chunks are viewed as symbols. However, if a chunk is completely contained within a transposed region, there is a chunk with matching checksum in some other relative position [Tri99] (see figure 23).

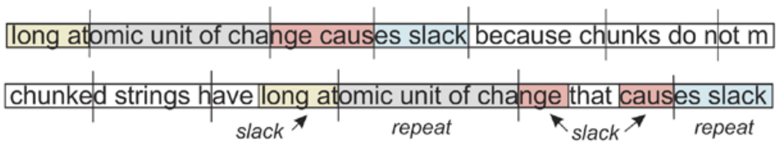

long atomic unit of change causes slack because chunks do not m
chunked strings have long atomic unit of change that causes slack
slack
repeat
slack
repeat

Figure 23: Chunked string with an alignment change and slack.

Chunk size affects compression performance, because transmitting a whole chunk implies transmitting some slack, which increases overhead [Tri99, BBG06]. Chunking granularity should always be much denser than the average change granularity to avoid excess slack [BBG06]. With smaller average chunk size, each chunk contains less slack on average, but there will be more chunks. For each chunk, some amount of metadata (fingerprint, chunk size) must be transmitted, and thus smaller chunks increase the metadata overhead [BBG06]. Larger chunks require less overhead space due to fewer chunks. When there is only small number of changes relative to string length and the changes are clustered close together, larger chunks require fewer bits to be transmitted. Number of chunks also affects computational requirements [Tri99]. For example, cryptographic hash functions often require initialization and padding per chunk [Tri99]. The average chunk size is an application-chosen compromise between slack and protocol overhead [BBG06]. Optimal chunk size is dependent on the string content and the characteristics of changes and no one chunk size is optimal for general case with arbitrary strings [IMS05, You06]. Chunk size could be optimized by sampling the input, but with secondary compression the chunk lengths are content-defined. Optimization would require knowing compressability of the chunks, which is a NP-hard problem [YIS08].

Tridgell [Tri99] notes that remote differencing is an asynchronous workload for the two hosts. In the worst case, reference host needs to send all of the fingerprints and version host still needs to send the complete version string. In the best case, version host does not need to send anything. Network speed is also often asynchronous: uplink speed from a mobile device can be slower by a factor of 100 than downlink speed to the mobile device, making any outgoing traffic unwanted. When the uplink speed is insufficient to transmit fingerprint data faster than downlink speed allows copying a compressed version string, differential compression should not be used at all [Tri99]. It is preferred that the mobile device transmit the least amount of data possible, whereas it would be preferred to have the server do least amount of computing possible and not need to have to maintain large session state to avoid malicious attacks by creating a large number of abandoned sessions.

4.4 Alignment Problem

Content-defined division points attempt to solve the alignment problem where adding or removing symbols shift the relative alignment of all symbols following the modification position, causing the checksums of all affected chunks to change. Bjorner, Blass and Gurevich [BBG06] make an assumption that when a symbol insertion or removal cause alignment mismatch between reference and version strings, there will soon follow a position where the strings would again be identical if one of the strings is re-aligned by repositioning the cutpoint function. If the same content-defined function is used for both strings, the function's output will again become identical after processing some small number of matching, again aligned symbols. A change should thus only affect few neighboring chunks, making content-defined chunking method s*hift-resistant*. A chunking method should also be *local*, that is only depend on current input and not need to keep a history [EsT05]. A local function can start anywhere within a string and after some initial input start to produce identical division points when compared to starting from the beginning. When chunk division is not dependent on previous content, the same substring will locally induce the same division points, even if the string is otherwise completely different. Content-defined chunking leads to variable-length chunks, because there is infinite number of possible different input string contents. No general chunking method can be optimized for all possible inputs [BBG06]. For example, if the cutpoint function is simply to insert a cutpoint whenever a certain symbol is encountered, but a string does not contain any such symbol, the cutpoint condition is never satisfied. The string could also consist of only such symbols, causing cutpoint condition to be satisfied on every position.

Chunk division points are defined by iterating through a string and evaluating the content with a chunk division function [BBG06]. When the content satisfies a given condition, a cutpoint is specified for the position. The cutpoint might not be set to the last evaluated position, but set to some previous position preceding or following the position, depending on the chunking function. Worst case inputs are not common in practice, but there can be long sequences of same symbol [BBG06]. Examples of problems are padding symbols in structured documents and areas of same image color values in picture documents. Bjorner, Blass and Gurevich [BBG06] suggest pre-processing with run-length encoding (RLE) prior to chunking to eliminate sequences of same symbol. However, if the differing chunks are compressed using secondary algorithm, long sequences and repeating patterns typically can be encoded into much shorter space, lessening the problem of long chunks. Unbounded chunk length would cause implementation issues such as overflowing variables or insufficient memory, so hard limits must usually be enforced. For practical applications, absolute guarantees are not crucial; it is enough that the probability of misbehavior is low [BBG06]. Avoiding unsuitable chunking parameters and thus reducing slack for different input strings would require analyzing the content before chunking. Both hosts should negotiate and agree on chunking parameters beforehand, which requires an additional communication round for transmitting the analyzation results. Otherwise the other host would not know how to perform chunking. We note that by selecting a tailored chunking method for a given string, it could be assumed that edits can be better contained within chunks, avoiding changes from crossing block boundaries so often.

Because string content is arbitrary, evaluating which chunk division function is better than other becomes a problem. Bjorner, Blass and Gurevich [BBG06] suggest measuring the variance of chunk size on random strings to compare different division functions, as the chunk size should be neither too small nor too large. The theoretical minimum of zero variance is only achievable when using constant length chunks [EsT05]. There cannot be a chunking method that guarantees a maximum length for chunks on all possible input strings, unless strict numeric length limit is enforced or unless some restricted input window is used for the division function [TBG06]. Strict minimum and maximum chunk size limits need to be enforced to avoid excessive or insufficient number of chunks or overflow of application variables. Enforcing chunk length limits interferes with chunks being content-defined by applying constant-length chunking to a subsection of a string [EsT05]. Deviating from being content-defined may interfere with finding alignment again. For example, if the current chunk was length-limited due to being too long, and there is a minimum length restriction for the next chunk, and the content-depedent cutpoint position would have been at the next symbol after enforced cutpoint, the alignment will be reached too late by the minimum amount [MCM01]. Variance calculations are done assuming random content, but in practice the assumption does not hold true due to e.g., repetitive patterns, and chunking algorithms produce worse results [EsT05].

Another method capable of detecting an unbounded number of both transposed and misaligned chunks was described Tridgell and McKerras [TrM96] and implemented as rsync tool by Tridgell. The basic idea of rsync algorithm is for one of the hosts to calculate fingerprints of fixed-length chunks from every possible position of its string and then compare each of the fingerprints against all of the fingerprints received from the other host. If there is a repeat corresponding to the chunk in some position of the other string, comparing against every position is guaranteed to find the repeat, even when it is transposed or partially overlapping with other chunks. The principle is same as with Obst's algorithm (discussed in section 3.2.1): chunks are the seed prefixes, and the fingerprints are received from reference host. However, now there are $n - c + 1$ fingerprints calculated on the version host, instead of approximately n/c. The fingerprint strength must be increased to $\log_2 n + \log_2(m/c) + b$ bits per chunk, to avoid increasing collisions probability. Content-defined chunking thus allows using $\log_2 c$ fewer bits per fingerprint than the two-level method. Comparing the chunk fingerprints against much smaller set of respective fingerprints instead of all positions also requires less computational resources [IMS05].

Another issue with comparing against all positions is that one host must compute $\Omega(n)$ fingerprints, which requires a lot of computational power. One of the hosts must calculate a factor of $c - 1$ more fingerprints; typically 700 times more [Tri99]. Tridgell and McKerras

[TrM96] solved the problem (and made the algorithm practical) by using a *two-level fingerprinting* scheme. The idea is to use more network bandwidth to allow computing less. The reference host calculates two separate fingerprints, both a weak and a strong, for each chunk the host has. The weak fingerprint is calculated using a fast rolling hash function and using fewer bits than is used for the strong fingerprint. The weak fingerprints are compared first, and only when there is a match, the strong fingerprint is computed and compared. Weak fingerprint is only used for optimization and its strength is only relevant to number of unnecessary strong fingerprint computations. Relatively few strong fingerprints need to be computed in the average case [BBG06]. We note that time saved by not calculating strong fingerprints should exceed time required to compute and transmit the weak checksums. Our empirical testing results (section 5.2) show this no longer holds true for modern fast enough multi-core processors and slow enough networks.

4.4.1 Content-Defined Division Functions

In this section we discuss different generic content-agnostic methods for applying content-defined chunking for arbitrary strings. Implementation-specific chunking methods relying on known content properties are more likely to produce better divisioning into chunks than generic methods. For all division methods, when the string is shorter than the minimum chunk length or when the function is never satisfied, the chunk size will be the length of the string.

Arbitrary strings may contain any permutation of symbols, which makes content-defined division problematic [BBG06]. The division functions discussed in this section use hash values calculated from a short sliding window. We assume uniform distribution of output values for hash function $h(x)$, thus hashing is expected to produce a more uniform distribution of values on any input string [BBG06]. Hashing is used to avoid some of the pathological cases, such as missing certain symbols (value ranges) from input strings. For example, an input missing half of the possible symbols is still expected to produce uniform fingerprint distribution, whereas any mask depending on the symbols themselves could fail due to not encountering any required symbols. Both the probability of hash collision and probability of not calculating a fingerprint with chosen properties at all are assumed to be low. Long sequences of identical symbol will still produce a long sequence of identical hash values, since the inputs to hash function are identical. Fingerprinting function’s sliding input window can be adjusted longer, to prevent short symbol repetitions causing repetitions of fingerprint values [BBG06].

Expecting a specific hash value would have too low probability of success [BBG06]. *Interval filter* as defined by Bjorner, Blass and Gurevich [BBG06] inserts a cutpoint after a contiguous sequence of u fingerprints from a given subset. The fingerprint value range is first divided into two disjoint subsets, X and C. Subset C contains the fingerprints used to detect chunk division positions. Fingerprint values from subset X are ignored. When iterating through a string, a position p is a cutpoint if $h(S_p) \in X$ and $u-1$ preceding fingerprints all belong to set C: $[h(S_{p-u-1}) \dots h(S_{p-1})] \in C, u \geq 1, p \geq u$. Probability of any p being a cutpoint on random input string is $(C/(C+X))^u$, which implies that u has to be small or C large relative to $C+X$. Average chunk length can be adjusted by adding or removing symbols from C and is expected to be eu [TBG06]. Minimum chunk length is p, bounded by the hash function's input window. There is no intrinsic maximum for chunk length. Variance of chunk length is $(1-(2u+1)x)/x^2$, where x is the probability of a position being division point. Minimal slack, acquired by choosing $(u+1)/(2u+1) = 1/2$ is approximately $3/4$, whereas expected slack is $1 - e^{-1} + e^{-2} \approx 0.77$ [TBG06].

Modulo arithmetic can be used to reduce the range of possible output values into suitable scale, increasing the probability of computing a specific hash value. The Low-Bandwidth File System [MCM01] implements a *point filter* method where a position p is a cutpoint when $h(S_p) mod\, q = x$, where q is a chosen modulus and x is a chosen constant. Assuming the fingerprint values are uniform, the probability of any position p being a cutpoint can be chosen by adjusting the modulus q. The probability of any S_p being a division point is $\frac{1}{|f|}\left\lceil\frac{|f|}{q}\right\rceil$. Variance of chunk length is approximately $q^2 - q$ [BBG06]. Expected slack is $1 - 1/q$, whereas minimum slack is 0.82 when $c = 0.3q$, where c is the minimum size limit of a chunk [TBG06]. The Low-Bandwidth File System calculates fingerprints with a Rabin rolling hash function using a sampling window of 48 bytes. Whenever $h(S_p)\; mod\; 2^{13} = 0$ (lowest 13 bits of fingerprint are zero), a division point is specified. Expected chunk size is $2^{13} + 48$ bytes, and the effect of sampling window's size to expected chunk size is small. For remote differencing, much shorter chunk length is practical. Long sequences of same symbol may prevent the division condition never to become true, and thus the method has no intrinsic minimum or maximum limit for chunk size [You06]. Division points that would produce chunks smaller than 2^{11} bytes are ignored and maximum chunk size is limited to 2^{16} bytes [MCM01]. The division function is not local [BBG06].

Instead of expecting some specific fingerprints, relative properties of fingerprint values can be tested. Schleimer, Wilkerson and Aiken [SWA03] introduced a method named *winnowing* suitable for chunk division, based on finding the local minimum of fingerprint values. Bjorner, Blass and Gurevich [BBG06] specified a similar algorithm, but instead based on finding the local maximum. The method sets division points based on unique maximum of a fingerprinting function $h(x)$ within a *local horizon*, which is defined as the substring $[S_{p-w} \dots S_{p+w}), p \geq w, p + w < n$. The position that gives the unique maximum value for the hash function within the horizon is chosen as division point (see figure 24). If there are multiple maximas, division point is not chosen. Each string position has an equal probability of $1/2w$ of being a cutpoint and the expected average chunk length is thus $2w$. The filter is symmetric, so the expected portion of slack is equal for both directions from the middle of a chunk, totaling ~0.7. Implementations typically specify w to magnitude of 2^8 and the number of possible different fingerprint values is in magnitude of 2^{32} to 2^{64}. Probability of having multiple shared maximas within a horizon is small, because $w \ll h(x)$. Probability of not having a local maximum at all is lower than with interval filter and point filter. Calculating the variance of the chunk length remains an open problem [BBG06]. Because the method considers only the unique maximum from a given horizon, it has an implicit minimum limit of $2w$ for the chunk size [TBG06].

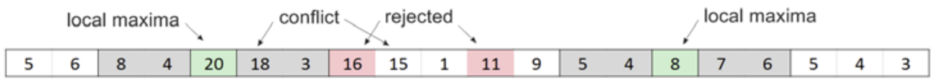


Figure 24: Local maximums of a string, using horizon size of $w=2$.

Local-maxima algorithm finds division points as follows. The pseudocode and description is based on our implementation (see figure 25), which is based on description by Bjorner, Blass and Gurivich [BBG06]. For clarity, we ignore some trivialities such as last block with length less than w, which is easily resolved using zero-padding. The string is divided into blocks of length w (28) or $w+1$ (7) and all of the blocks are processed in order, from left to right (7, 28). Within a block, all positions i are processed in reverse order, from right to left (9, 30). The algorithm stores a *greedy sequence* containing all positions i where $h(i)$ is greater than all previous found $h(v), v > i$ within the currently active block and thus also the greedy sequence (13, 36). Values in greedy sequence are thus in increasing order of h and the last value is the largest. The last value in the sequence is a candidate for division point, except when another equal fingerprint is found – then the candidate status is cleared and the duplicate value is not added into the sequence (15, 38). There are two kinds of blocks, those without an active candidate ('first half') and those with a candidate ('second half'). When a 'first half' block has been processed, if a candidate was found, there should not be another equal or larger value within w symbols in either direction. To reduce the number of comparison, instead of scanning backward to check, the algorithm uses the greedy sequence to check for equal or larger values from immediately preceding block (17…22). When a larger value is found, the candidate status from current block is cleared (21), and next block is scanned for candidates (6…16), otherwise next block ('second half') is scanned without clearing the candidate (26…39). After the scanning, if no larger values were found (32), the active candidate from preceding block is chosen as division point (46), as no value from current block was greater, and the preceding block has already been checked (otherwise the candidate status would have been cleared). Processing moves to next block (6). If a larger value was found, it will become an active candidate, and next block is scanned (26).

The greedy sequence is formed in $O(n-1)$ time by iterating through every position of a block and contains an expected number of $\log_2 w$ values. Checking for conflicting values from preceding block thus requires $\log_2 w$ comparison on average, instead of w. Thus, chunking of a string requires $O(n + (n/w)\log_2 w)$ time on average. Worst case time bound is $O(2n)$ and space bound is $O(2w)$.

```
Algorithm LocalMaximaChunkDivisor (S,w)
maximas ← array()
precedingMaximas ← array()
maximas.append(0, 0)
pos ← 0
while (pos < S.length) {
  MaximaFirstHalf: {
    SwapHalves(w + 1)
    index ← pos - 1
    while (index-- >= 0) {
      hash ← hashFunc(S + index)
      if (hash >= maxim.hash) {
        if (hash > maxim.hash)
          maxim ← maximas.append(hash, index); dupeMaxim ← false
        else
          dupeMaxim ← true
      } }
    if (dupeMaxim = false) {
      foreach (precedingMaximas as pm) {
        if (pm.index < maxim.index - w) break
        if (pm.hash >= maxim.hash) {
          maxim ← null
          break
        } } }
    if (maxim = null) continue MaximaFirstHalf
  }
  MaximaSecondHalf: {
    divisionPointCandidate ← maximas.last()
    SwapHalves(w)
    index ← pos - 1
    while (index-- >= 0) {
      hash ← hashFunc(S + index)
      if (hash >= precedingMaximas.last().hash && index <= precedingMaximas.last().index)
        divisionPointCandidate = null
      if (hash >= maxim.hash) {
        if (hash > maxim.hash)
          maxim ← maximas.append(hash, index); dupeMaxim ← false
        else
          dupeMaxim ← true
      } }
    If (dupeMaxim = true)
      maxim ← null; dupeMaxim ← false
    if (divisionPointCandidate = null) {
      if (precedingMaxima = maxim.hash) continue MaximaFirstHalf
      else continue MaximaSecondHalf
    } else {
      onChunkDivisionPoint(divisionPointCandidate.index)
      continue MaximaFirstHalf
    } } }

function SwapHalves (width) {
  swap(maximas, precedingMaximas)
  maximas.clear()
  maxim ← maximas.append(0, 0)
  precedingblockLen ← blockLen
  blockLen ← width
  pos ← pos + blockLen
}
```

Figure 25: Pseudocode for local maxima chunk division function.

To solve the problem of too short or too long chunks, Eshghi and Tang [EsT05] present an algorithm named Two Thresholds, Two Divisors that uses a secondary chunking method for situations when the primary method produces too small or too large chunks. To avoid too large chunks, the secondary chunking method uses smaller modulus divisor or shorter w for its division function. Producing hash values from a smaller range of values causes finding division points more often. The algorithm still implements a hard maximum size limit for the secondary function, using 2.8 times of the average chunk size as the limit. Bjorner, Blass and Gurevich [BBG06] suggest ignoring a division point when chunk size is less than desired, thus joining the short chunk into the following chunk. Sheng, Xu and Wang [SXW10] suggest using fixed-length chunking as second-level division method.

The local-maxima filter was found to cause less slack and checksum overhead in practice than interval filter and fingerprint mask filter in testing done by Bjorner, Blass and Gurivich [BBG06]. In practice a naive fingerprinting function (not actually calculating a hash at all) is faster in execution time than Rabin's polynomial hash function and still sufficient in finding locally maximal fingerprint [TBG06]. We however noted in our empirical testing that alignment changes (string modifications) cause second-order changes in chunk division points due to the local maximas moving cross horizon boundaries. When sequences of hash values are identical in two strings but horizon boundaries are shifted, some unchanged chunks are split or joined and thus can no longer match. We note the local-maxima function could be used as position selection function for sparse fingerprinting in local differencing. Local-mimima would have identical performance due to symmetry. However, when comparing content directly without hashing, local maximum would likely yield better performance, as zero-padding is common in practice.

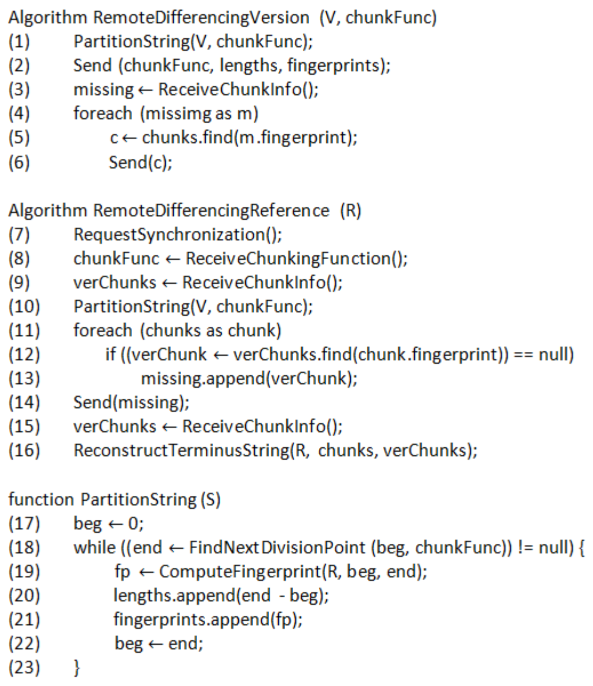
4.5 Differencing Fingerprints

We now discuss algorithms for inferring which chunks of version host are not present on reference host and which chunks have been transposed. In subsection 1 we discuss differencing algorithms based on (recursive) comparing of chunk fingerprints. The general idea is that one host transmits a set of fingerprints, and the other host responds with comparison results. These algorithms share a communication space bound of $O(n \log n)$. In subsection 2 we discuss another approach to differencing based interpolating solutions to a group of polynomial equations. A set of evaluations of a characteristic polynomial of the fingerprints is communicated from reference host to version host. No fingerprints are sent for differencing. On the version host, factoring is used to reconstruct the polynomial, which can be used to infer which fingerprints are present on the reference host. Communication space bound of the algorithm is *not* dependent on the length of the strings and is $O(2d)$, where d is the number of differing chunks. Both methods perform the same task of inferring which fingerprints are missing from reference host or relocated, and thus these two methods are interchangeable as part of an implementation [YIS08].

4.5.1 Fingerprint Comparison

Remote differencing algorithms can be divided into two classes depending on whether multiple communication rounds are allowed or not: multiple-round and single-round. A communication round consists of one host transmitting a message and another host responding. Additional communication rounds allow communicating fewer bits in total, as after each message the hosts have more information allowing avoiding sending information known to be unnecessary [YIS08]. Each round requires waiting for the network's round-trip time. High communication latency is a problem because while waiting for response from other host, large amounts of data could be transmitted. For example, using a synchronous wireless network capable of transmitting 8 mebibits per second and having a round-trip-time of 100 ms, ~100 kibibytes could be transmitted from host to another in time the first byte is communicated back to originating host. Latency can make single-round algorithms more efficient for synchronizing singular short strings [YIS08]. Multi-round communication also requires the other host to maintain session state, which can be problematic due to malicious users and resource limitations [IMS05]. For example, the session state includes data structures storing information about remaining unmatched chunks, such as a bitmap [SNT04]. Another reason preventing multiple communication rounds is maintaining compatibility with existing applications or protocols that do not allow maintaining session state [MKD02, IMS05].

Fingerprint differencing with single communication round is performed as follows, based on algorithm by Tridgell and McKerras [TrM96] (see figure 26). A similar algorithm was described by Pyne [Pyn95]. The reference host first notifies version host (7). Version host divides its string into chunks and calculates a fingerprint for each chunk (1). Version host then transmits all of the fingerprints to the reference host using a single communication (2). The communication includes parameters of chunk division function and chunk sizes. Size list allows inferring chunk positions. Each chunk can be identified by its index within the list. Reference host stores all of the fingerprints into a hash table (9). Reference host will then iterate through its string and divides the string into chunks using chunk division function identical to what the version host used (10). A fingerprint is calculated for every chunk (19). Each fingerprint from reference string is queried from the hash table (11…12), and thus compared against all fingerprints from version string, which allows finding transposed chunks. When the fingerprints do match, the algorithm assumes that the chunk contents are identical and proceeds to the next reference string fingerprint (11). When the fingerprint does not match, the algorithm inserts the index of the fingerprint into a list of missing chunks (13). After all version fingerprints have been checked, the missing chunks list contains indices of all those version host fingerprints that the reference host does not have. Reference host requests the missing chunks' contents by transmitting the fingerprints to version host (14). Version host responds by sending the chunk substrings (4…6), and algorithm is completed. The lower bound of communication for this algorithm is $\Omega(m/c)$ [Tri99], space bound is $O(1)$ and time bound is $O(m)$.



```
Algorithm RemoteDifferencingVersion  (V, chunkFunc)
(1)     PartitionString(V, chunkFunc);
(2)     Send (chunkFunc, lengths, fingerprints);
(3)     missing ← ReceiveChunkInfo();
(4)     foreach  (missimg as m)
(5)         c ← chunks.find(m.fingerprint);
(6)         Send(c);

Algorithm RemoteDifferencingReference  (R)
(7)     RequestSynchronization();
(8)     chunkFunc ← ReceiveChunkingFunction();
(9)     verChunks ← ReceiveChunkInfo();
(10)    PartitionString(V, chunkFunc);
(11)    foreach  (chunks as chunk)
(12)        if ((verChunk ← verChunks.find(chunk.fingerprint)) == null)
(13)            missing.append(verChunk);
(14)    Send(missing);
(15)    verChunks ← ReceiveChunkInfo();
(16)    ReconstructTerminusString(R,  chunks, verChunks);

function PartitionString (S)
(17)    beg ← 0;
(18)    while ((end ← FindNextDivisionPoint (beg, chunkFunc)) != null) {
(19)        fp  ← ComputeFingerprint(R, beg, end);
(20)        lengths.append(end  - beg);
(21)        fingerprints.append(fp);
(22)        beg ← end;
(23)    }
```

Figure 26: Pseudocode for comparison-based fingerprint differencing.

To solve the alignment problem by comparing reference chunks against every position of version string, the algorithm can be modified to use the two-level fingerprinting as follows, as described by Tridgell and Mackerras [TrM96]. The reference host calculates both a weak and a strong fingerprint for each constant-length chunk, instead of just the strong fingerprint. The weak checksum is calculated using a fast rolling hash function using fewer bits, while the strong checksum is calculated using a cryptographic hash function using the same number of bits than without two-level fingerprinting. Both sets of fingerprints are then sent to version host. Version host stores the two sets of fingerprints into two separate hash tables, instead of storing strong fingerprints into a hash table using weak fingerprints as keys. Weak hashes may cause collisions, but weak hashes are only used as a probabilistic filter. Boolean membership queries are sufficient, and hash chaining is not needed. Version host then iterates through its string and calculates a weak fingerprint for all intervals of length w from every position. Each weak fingerprint is queried from the hash table and when a fingerprint is found to exist, only then a strong fingerprint is calculated and compared for verification. After matching a chunk, weak hash comparison is continued at the end position of the matched repeat (overlapping repeats are not considered).

While two-level fingerprinting allows finding matches when there are alignment changes or long enough transposed substrings (at least one whole chunk within the transposed substring), it requires transmitting the weak fingerprints and requires larger strong hashes. Each chunk requires $\Theta((h_{weak} + \log_2 c)(n/c))$ bits of additional data (c is the average chunk size). With Tridgell's and McKerras' [TrM96] implementation, the weak fingerprints are calculated using a function based on Adler-32, which is based on Fletcher's checksum. Strong fingerprints are calculated using 128-bit MD5, thus requiring a total of 160 fingerprint bits per chunk [TrM96, Tri08]. The two-level scheme is effective only when there are few fingerprint collisions between the weak fingerprints. Choosing the weak hash function is a compromise between speed and collision probability [Tri99]. Differencing against a string consisting of a long sequence of same symbol causes all positions to match the same weak fingerprint, thus causing the version host to calculate strong fingerprints for every position of a string [BBG06]. Worst case execution time is thus worse by factor of c than without the modification. A denial of service attack can easily be launched against a public server by requesting synchronization of a worst-case string containing only long sequence of same symbol, causing all weak checksums to match [BBG06].

Roles for computing either weak fingerprints for chunks or computing for every position are interchangeable. Depending on the implementation, version host can be the one calculating and comparing fingerprints for all positions [Tri99]. Decision on which host will have which role is based on upstream network bandwidth and security considerations [Tri99]. Reference host needs to do more computation and version host needs to send all checksums, regardless if there are any differing chunks. Often the server should do the least computation. We note that the two-level modification can be used with variable-length chunks by choosing the shortest of the lengths as the upper bound for the size w of input window for the rolling hash function (using shorter w is also possible). When a chunk is longer than w, the remaining suffix of the chunk is ignored and excluded from input of the weak hash function. Thus, the weak hash no longer represents a chunk, but instead a seed prefix of a chunk, which can span up to the whole chunk. We also note that the two-level algorithm is similar to dictionary searching for local differencing. Instead of performing a symbol-wise comparison, Obst's algorithm could compute and compare strong fingerprints. Instead of having string positions in the hash table, it could contain strong hashes. An algorithm could thus avoid any random access related to verification comparison.

The described single-round algorithm requires all of the reference checksums to be sent, even when the strings are identical (the hosts do not know that), which adds overhead to the protocol [Tri99]. When synchronizing a long string there can be millions of chunks. There are often only few changes made to the string and most of the chunks (and fingerprints) are identical between the strings [IMS05]. Compression efficiency can be increased by allowing multiple communication rounds between hosts so that they can avoid transmitting some of the fingerprints by exchanging information progressively [TBG06]. Multi-round *recursive chunking* reduces the total number of fingerprints needed to be transmitted by first starting with *superchunks* representing multiple chunks and then recursively dividing the chunks again into progressively smaller chunks on each communication round (see figure 27) [TBG06]. When two sequences of fingerprints are identical on both host, the *superfingerprints* are also identical, and the algorithm can avoid transmitting the lower level fingerprints. Recusive chunking requires hosts to calculate multiple levels of fingerprints, which can be done using a single reading pass, but requires more computation proportionally to levels of recursion. Requiring multiple levels of fingerprints also delay startup of transmitting, because higher level fingerprints cannot be computed until there are sufficient number of lower level fingerprints. Improvements in compression ratio can mask the additional latency and protocol overhead. Multi-round synchronization can reduce bandwidth usage by a factor of 2 to 3 [Lan01, IMS05, SNT04], when compared to the generic single-round comparison algorithm. The longer the string, the more multiple-round communication can improve synchronization time [TBG06].

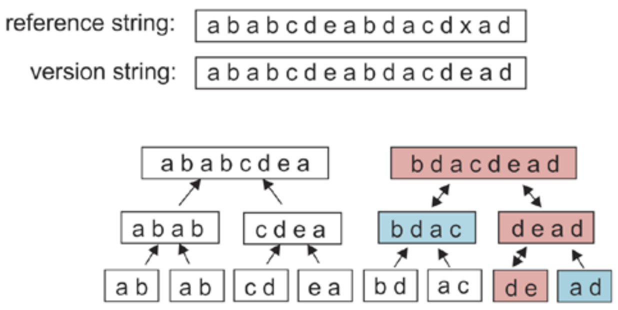


Figure 27: Recursive comparison-based fingerprint differencing.

A synchronizer might begin with a superchunk size of 2^{20} bytes and then divide the chunk size into half on each communication round, until finally having a chunk size of 2^8 bytes after 11 rounds. For comparison, Tridgell's popular single-round rsync algorithm uses chunk size of $\sqrt{n}$ bytes with a minimum limit of 700 bytes [Tri99, You06]. Best case allows detecting that the whole string is identical on the first round, thus requiring transmitting only a few superfingerprints [SNT04, You06]. Worst case of having two completely different strings is worse than with single-round synchronization, because $O(\log_2 C)$ additional units of data had to be transmitted using and $O(\log_2 |C|)$ communication rounds to detect that all of the fingerprints were different. The chunk size when recursion should be stopped is ~$2^6 \ldots 2^7$ bytes, which is when the increase in fingerprint data exceeds the decrease in slack [SNT04]. The upper bounds for transmit when splitting chunks recursively is $O(k\,(\log_2 n)^2)$, but this does not apply when block copying is allowed [IMS05].

Recursion using constant-length chunks again exhibits the alignment problem: adding or removing enough symbols to insert or remove a chunk from a string, the contents of all superchunks following the modification position will be different [BBG06]. All superfingerprints for higher recursion levels change. Removing the first lowest-level chunk would change all superfingerprints. Because multiple levels of fingerprints are calculated before transmitting anything to the other host, comparison against all string positions cannot circumvent the alignment problem [BBG06]. Teodosiu et al. [TBG06] presented a solution that recursively applies content-defined chunking on the fingerprints. A string is chunked and fingerprints are calculated from the chunks. All of the fingerprints are then catenated into a temporary string. The temporary fingerprint string is chunked using content-defined variable-length chunking function, thus providing a new smaller set of superfingerprints, having $(\log n)^c$ fingerprints. Each new superfingerprint now represents a variable-length sequence of chunks of the original string. The new set of superfingerprints is again recursively catenated into another temporary string and fingerprinted, until a sufficiently short temporary string is produced. This method first synchronizes the temporary superfingerprint strings, starting from the highest level, and then uses a final communication round to synchronize the actual strings.

Optimal number of recursions is dependent on network latency and bandwidth: too short first temporary string for the superfingerprints can lead to one additional communication round and the latency exceeds the time saved from transmitting less data. Based on their empirical testing, Bjorner, Blass and Gurevich [BBG06] recommend using a smaller chunk size of 2^8 for the superfingerprints and 2^{11} for the strings. There is no need for all of the recursive chunking rounds to use the same chunking method. For example, two-phase chunking [SXW10] can use different chunking method for later rounds. We note that because each communication round requires waiting the round-trip latency, the recursion should be optimized so that the first message contains at least a full protocol layer packet's length of fingerprints. Using an estimate of bandwidth corresponding to the round-trip latency, the algorithm can calculate whether it is faster to transmit more fingerprints within one message, or split the message by introducing one additional level of recursion.

4.5.2 Charasteristic Polynomial Interpolation

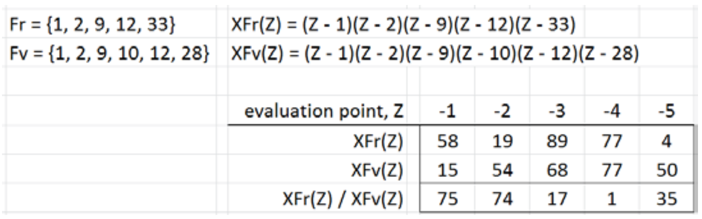
The fingerprints can be viewed as an unordered multi-set of integers, $F = \{x_0, x_1, \dots, x_{n-1}\}$, if the order of the fingerprint sequence is given separately. Consequently, the fingerprint differencing problem can be viewed as a set reconciliation problem of unordered integers. Solving the problem will reveal which fingerprints are missing, but does not reveal transpositions or to which position the missing fingerprints belong to. Starobinski, Trachtenberger and Agarwal [STA03] note that it is possible to reconstruct any subset of missing fingerprints of size d by solving a set of polynomial equations, instead of transmitting fingerprints for comparisons. The set reconciliation can be solved with a communication space bound of $\mathcal{O}(2d)$, where d is an upper bound for the number of symmetric differences [STA03]. Symmetric difference means that if one fingerprint is changed on version host, it is then missing from reference host, but the unchanged fingerprint on reference host is now also missing from version host. One change thus equals two differences. The required bandwidth is not dependend on length of the strings (number of fingerprints), but only on allowed maximum number of differences. Set reconciliation algorithm however requires prior knowledge for upper bound of d, which means that an algorithm has to choose d [STA03]. Also, set reconciliation considers only the content of the elements, not their relative ordering, which requires transmitting additional information [STA03]. The amount of additional information is constant per fingerprint and thus does not change the asymptotic bound.

The polynomial is not a summary function like a hash function is because it contains all the information in the set [MTZ03]. Minsky, Trachtenberg and Zippel [MTZ03] describe the operating principle of the method as follows. When dividing a polynomial with another, the common terms cancel each other out. The polynomial itself cannot be transmitted with fewer bits than the set, but it is possible to compute values from a small number of evaluation points $e \geq k$ (where k is the number of actual differences, not the upper bound d). The results of the divisions can be used to interpolate the rational function by factoring $\mathcal{X}F_r$. Let $\Delta_v = F_v \setminus F_r$ and $\Delta_r = F_r \setminus F_v$. All terms that correspond to elements in both Δ_r and Δ_v will cancel each other out, leaving only $\mathcal{X}\Delta_r(Z)$ and $\mathcal{X}\Delta_v(Z)$ (see figure 28). The idea is to divide out the terms of the polynomials at a set of evaluation points. When Δ_r and Δ_v are small, the interpolated rational function has a small degree, requiring a small number of evaluation points. However, this implies that both Δ_r and Δ_v are known, which they are not – each host knows only its own polynomial. Choosing the number of evaluation points becomes a problem. Sending too few sampled evaluations by choosing a d smaller than number of differing fingerprints prevents interpolating the rational function because there are multiple possible answers [STA03]. The result of the division is a parity sum of the difference set. When there is only a single error, the division produces the missing bit sequence [MTZ03].

$$\frac{\mathcal{X}F_r}{\mathcal{X}F_v} = \frac{\mathcal{X}F_r \cap F_v(Z) \cdot \mathcal{X}\Delta_r(Z)}{\mathcal{X}F_r \cap F_v(Z) \cdot \mathcal{X}\Delta_v(Z)} = \frac{\mathcal{X}\Delta_r(Z)}{\mathcal{X}\Delta_v(Z)}$$

Figure 28: Common terms of charasteristic polynomials cancel each other out.

Starobinski, Trachtenberger and Agarwal [STA03] describe an algorithm for synchronizing unordered chunks. Both hosts start by dividing their string into chunks, separately on each host, using some identical division function. Each host then computes a strong fingerprint for each chunk and uses the fingerprints to compute characteristic univariate polynomials $\mathcal{X}F(Z) = (Z - x_0)(Z - x_1) \dots (Z - x_{n-1})$ for the set of fingerprints. Each host computes their own characteristic polynomial independently. The hosts choose $2d$ evaluation points each and then evaluate the polynomial at the chosen sample points (see figure 29). Both hosts need to have the same evaluation points. The evaluation points cannot be the same as any fingerprint, because a characteristic polynomial would then evaluate as zero. A seed number can be transmitted from one host to another to enable generating identical pseudo-random points, instead of sending a series of numbers [YIS08]. Version host transmits the evaluation results to the reference host, after which version host does not need to do more computation. Reference host builds a Vandermonde matrix and solves it using back substitution, retrieving a set of coefficients that are converted to polynomials. It then calculates the greatest common divisor of both hosts' polynomials and divides them to eliminate common terms (fingerprints). Then, reference host evaluates the polynomials to check that the results are valid. Finally, reference host uses factoring to solve the roots of the polynomials. The solved roots are the missing fingerprints: numerator gives the fingerprints missing from version host, and denominator the fingerprints missing from reference host. After solving which fingerprints reference host is missing, version host iterates through all its fingerprints and transmits all chunks substrings that the reference host does not have. The algorithm does not reveal where the differing chunks (fingerprints) belong to or if any chunks are transposed. Because communication cost is not dependent on number of chunks, the average chunk length can be shorter than with comparison-based algorithms, thus reducing slack [MTZ03]. The same message could be broadcasted to multiple reference hosts, and each one could independently compute the missing fingerprints, as long as number of symmetric differences to version fingerprint set is smaller than d on each host [MTZ03].



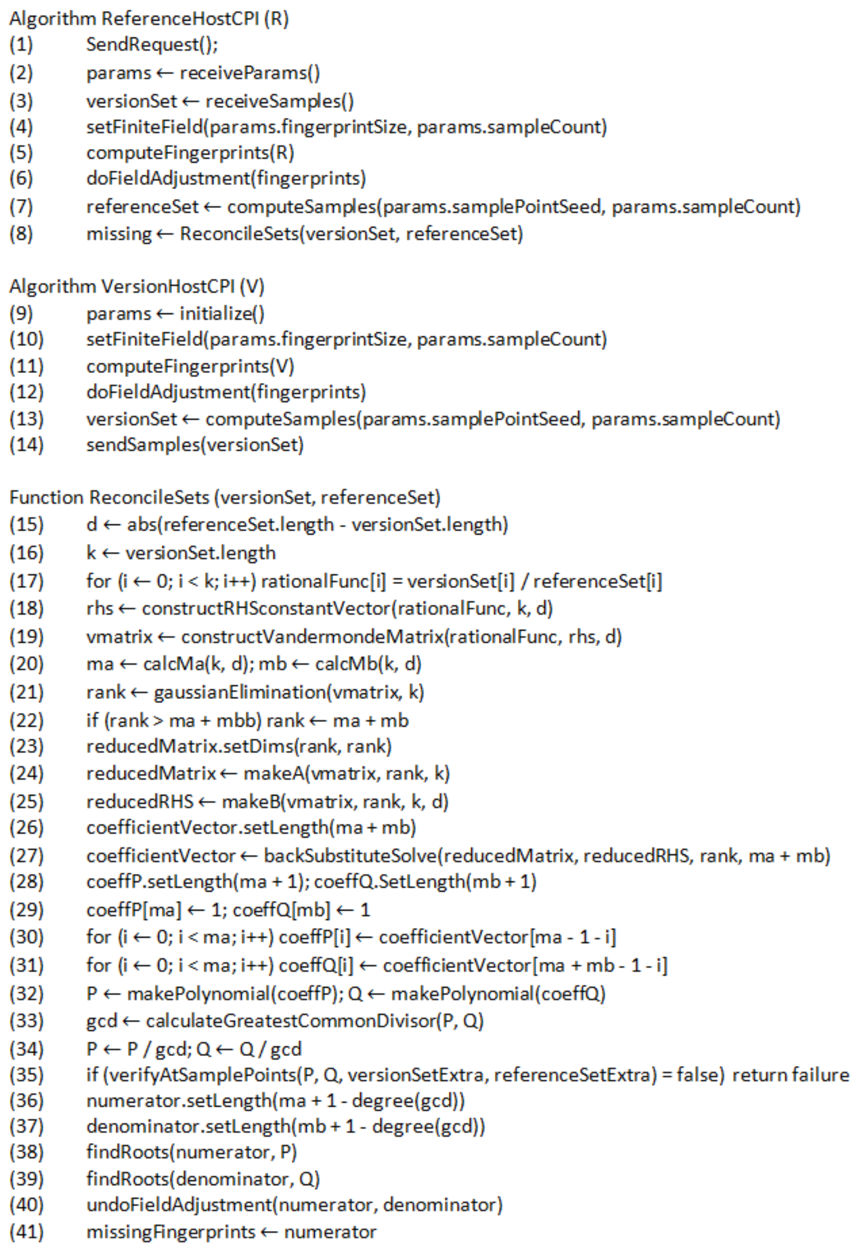
Fr = {1, 2, 9, 12, 33}	XFr(Z) = (Z - 1)(Z - 2)(Z - 9)(Z - 12)(Z - 33)					
Fv = {1, 2, 9, 10, 12, 28}	XFv(Z) = (Z - 1)(Z - 2)(Z - 9)(Z - 10)(Z - 12)(Z - 28)					
	evaluation point, Z	-1	-2	-3	-4	-5
	XFr(Z)	58	19	89	77	4
	XFv(Z)	15	54	68	77	50
	XFr(Z) / XFv(Z)	75	74	17	1	35

Figure 29: Sample values calculated over finite field $\mathbb{F}_{97}$ from chosen evaluation points [MTZ03].

For detecting positions and transpositions of chunks, Chauhan and Trachtenberg [ChT04] describe a modified version of the set reconciliation algorithm. To convert the ordered set of chunks unordered, they used fixed-length chunks starting at every position of a string and that overlap the next chunk by $c - 1$ symbols. The idea is to use the overlapping interval for inferring where the chunk belongs to, like pieces of a puzzle [ACT06]. When the overlapping suffix matches a prefix of another chunk, the chunks are assumed to be successive. It is possible to use a bitmask filter to create a non-contiguous overlap by including those following symbols that have the corresponding mask bit set. The hosts construct modified de Bruijn directed graphs from their chunks. Because the graphs may allow finding multiple Eulerian cycles, and only some of them correspond to the string, the cycles are enumerated. The ordinal of the correct Eulerian cycle is sent to other host, thus requiring only single integer. To solve the problem of converting a multi-set into a set, each chunk is appended with the number of times it has occurred in the multi-set, before fingerprinting the chunk [MTZ03]. Because the chunks overlap, single change may affect multiple chunks [MTZ03]. Each symbol position corresponds to c chunks, increasing the number of differences k proportional to chunk size c [YIS08]. The algorithm requires $O(k(\log n)^2)$ bits on average with two communication rounds, when the effective mask length is $\log n$ [YIS08].

Yan, Irmak and Suel [YIS08] further refined the algorithm by using variable-length content-defined chunk division and by complementing the message to other host with explicit ordering information. To eliminate ambiquities, the overlapping interval has to be long. Transmitting explicit position information to resolve ambiquities allows using shorter overlap length. Each chunk (except the last one) is extended forward by b symbols to create an overlapping interval for successive chunks, but no de Bruijn graph is constructed. Choosing the overlap length is a compromise between increasing number of differences and increasing number of ambiguous postions, requiring explicit position information. Increasing the overlapping length reduces the number of different possible positions for a chunk, also making it possible to encode the sequence of differences with fewer bits [YIS08]. The algorithm requires $O(k \log_2 n)$ bits on average, using a single communication round [YIS08]. The transmission includes the number of chunks version host has, and chunk substrings and their lengths. Additionally, if there are any chunks that have ambiguous positions based on overlapping, an enumeration ordinal of the correct choice [YIS08]. Ordering info is required only for those chunks whose position cannot be inferred by the overlapping interval, requiring $\lceil \log_2 y \rceil$ bits, where y is the number of possible chunks following [YIS08]. Secondary compression can be used for the substrings. It is possible for a version host to reconciliate sets with several different reference hosts using the same transmission content, because same information is required to be transmitted regardless of how the reference string is different [YIS08].

We now present the pseudocode for characteristic polynomial interpolation based set reconciliation, as described by Starobinski, Trachtenberger and Agarwal [STA03] (see figure 30). For clarity, we have omitted special cases, such as when either set of fingerprints is empty or shorter than chosen d, or when lengths of fingerprint sets differ by more than d. A reference host first requests samples from version host (1). Version host initializes its parameters based on chosen fingerprint size and maximum number of differences (9…10). Finite field large enough to hold fingerprints and evaluation points is used to limit the size of the product of polynomial factors. Version host then divides its string into chunks using some division function and computes fingerprints for each chunk (11). Out of the set of fingerprints, version host chooses a number of evaluation points and computes the sample values, which are then sent to reference host (13…14). After reference host receives the request parameters and version samples (2…3), it initializes itself with same parameters (4) and computes fingerprints from its string the same way reference host did (5). Reference host evaluates the polynomial at the same evaluation points as version host did (7). To make room for symmetric evaluation points, the range of fingerprint values is shifted on both hosts (6, 12). The evaluation points are chosen as d largest and smallest values of the finite field. Of the samples, reference host constructs a Vandermonde matrix (15…19), which is then reduced to smaller dimension using Gaussian elimination (20…25). The algorithm then solves the resulting group of equations using back substitution (26…31), and calculates the greatest common divisor (32…33) to eliminate common terms (34). The result is then verified by evaluating the polynomials at the evaluation points and comparing results (35). Additional sample values can be used for verification. In case the verification fails, there are too many differences in the fingerprint sets and set reconciliation must be retried using more samples. When the verification matched, the algorithm can use factorization to solve the roots for the numerator and denominator of the characteristic polynomial equation (36…39). The roots of numerator are the fingerprint values missing from reference host, and it can request the contents of respective chunks from version host. To detect transpositions and duplicate chunks, the fingerprints should be altered so that the position of the chunk affects the fingerprint calculation, in addition to just the chunk content. Version host must also transmit chunk positions, because it cannot be inferred implicitly as can be done with recursive fingerprint comparison. The roots of denominator are the fingerprint values missing from version host, but the information is not required for one-way synchronization. Roles of version host and reference host can be switched, so that version host does the factorization. The change allows synchronization with single request-response round, but may prevent version host from servicing multiple reference hosts because the computation requirement is high.



```
Algorithm ReferenceHostCPI (R)
(1)     SendRequest();
(2)     params ← receiveParams()
(3)     versionSet ← receiveSamples()
(4)     setFiniteField(params.fingerprintSize, params.sampleCount)
(5)     computeFingerprints(R)
(6)     doFieldAdjustment(fingerprints)
(7)     referenceSet ← computeSamples(params.samplePointSeed, params.sampleCount)
(8)     missing ← ReconcileSets(versionSet, referenceSet)

Algorithm VersionHostCPI (V)
(9)     params ← initialize()
(10)    setFiniteField(params.fingerprintSize, params.sampleCount)
(11)    computeFingerprints(V)
(12)    doFieldAdjustment(fingerprints)
(13)    versionSet ← computeSamples(params.samplePointSeed, params.sampleCount)
(14)    sendSamples(versionSet)

Function ReconcileSets (versionSet, referenceSet)
(15)    d ← abs(referenceSet.length - versionSet.length)
(16)    k ← versionSet.length
(17)    for (i ← 0; i < k; i++) rationalFunc[i] = versionSet[i] / referenceSet[i]
(18)    rhs ← constructRHSconstantVector(rationalFunc, k, d)
(19)    vmatrix ← constructVandermondeMatrix(rationalFunc, rhs, d)
(20)    ma ← calcMa(k, d); mb ← calcMb(k, d)
(21)    rank ← gaussianElimination(vmatrix, k)
(22)    if (rank > ma + mbb) rank ← ma + mb
(23)    reducedMatrix.setDims(rank, rank)
(24)    reducedMatrix ← makeA(vmatrix, rank, k)
(25)    reducedRHS ← makeB(vmatrix, rank, k, d)
(26)    coefficientVector.setLength(ma + mb)
(27)    coefficientVector ← backSubstituteSolve(reducedMatrix, reducedRHS, rank, ma + mb)
(28)    coeffP.setLength(ma + 1); coeffQ.SetLength(mb + 1)
(29)    coeffP[ma] ← 1; coeffQ[mb] ← 1
(30)    for (i ← 0; i < ma; i++) coeffP[i] ← coefficientVector[ma - 1 - i]
(31)    for (i ← 0; i < ma; i++) coeffQ[i] ← coefficientVector[ma + mb - 1 - i]
(32)    P ← makePolynomial(coeffP); Q ← makePolynomial(coeffQ)
(33)    gcd ← calculateGreatestCommonDivisor(P, Q)
(34)    P ← P / gcd; Q ← Q / gcd
(35)    if (verifyAtSamplePoints(P, Q, versionSetExtra, referenceSetExtra) = false) return failure
(36)    numerator.setLength(ma + 1 - degree(gcd))
(37)    denominator.setLength(mb + 1 - degree(gcd))
(38)    findRoots(numerator, P)
(39)    findRoots(denominator, Q)
(40)    undoFieldAdjustment(numerator, denominator)
(41)    missingFingerprints ← numerator
```

Figure 30: Pseudocode for set reconciliation based fingerprint differencing.

Computing the values at the evaluation points requires evaluating both strings completely, requiring $O((n+m)d)$ time. On the reference host, the interpolation problem is solved by solving $2d+1$ linear equations. Factoring the linear equations using Gaussian elimination requires $O(d^3)$ time. Because integers larger than processor machine word are used (strong fingerprints), additional processing is required per operation due to need to compute in pieces using smaller integers. Trachtenberg and Zippel [MTZ03] note that there are asymptotically faster algorithms, but it is unclear whether they are faster in practice. Validating the rational functions can be done probabilistically by evaluating them at additional random points and comparing that the results are the same. Communication requires $O((b+1)d)$ bits without additional evaluation samples for testing the rational functions. Solving the required upper bound d for differences could be done by iteratively attempting to reconcile and then retry with transmitting progressively larger number of sample evaluations after each failure. Testing can be done so that d is increased each time the test fails, leading to $O(d^4)$ execution complexity (as the test can fail for each d), or so that the number of test evaluations is increased by factor of c after each failure, leading to $O((d+c)^3)$ complexity when c is constant.

Set reconciliation is not suitable for use cases where number of differences is large, due to cubic or quartic time bound. Choosing d as close as possible to the correct number of differences is important, because overestimating causes unnecessary computation and bandwidth usage, and underestimating causes a failure of the algorithm or requires another iteration with larger d [YIS08]. Knowing the magnitude of d would allow switching to recursive comparison algorithm when d is too large [YIS08]. Recursive fingerprint comparison becomes faster approximately when $d \geq \log n$ [YIS08]. The problem however is that the number of differences is not known beforehand and failure cannot be detected before getting the results from the factorization. Esimating upper bound for d can be done by sending partial fingerprints having $(\log|samples|) + 8$ bits per sample, and infer from the ratio of matches an estimate for total number of differences [AgT06, YIS08]. When d is chosen as the actual number of differences, the method is nearly communication optimal [MTZ03]. Finally, as noted earlier, synchronization is related to transmitting with error correcting codes. The data sent by this algorithm can be viewed as redundancy data of a transformed Reed-Solomon error correcting code [YIS08].

4.6 Synchronization

Similar to local differential compression, remote synchronization is done in three phases [TrM96]. First, the hosts use some remote differencing algorithm to infer which chunks are missing or relocated from the reference host. In the second phase, version host transmits chunk position information and the contents of missing chunks from version host to reference host. Position information is needed for handling chunk transpositions and duplicate chunks. A differencing algorithm that cannot detect transpositions does not need to transmit position information, (implicit) chunk index information is sufficient. For matching chunks, version host transmits the chunk identifier and chunk position, which is analoguous to a copy directive for delta encoding. For non-matching chunks, version host transmits the chunk contents, length and position, which is analoguous to an insert directive. The insert directives' substrings can be compressed using a secondary compression algorithm, having compression contexts that refer to including surrounding substring that are not transmitted [SNT04]. From the set of copy and insert directives, a reconstruction algorithm can infer which intervals have been removed and explicit remove directives are not required.

Reconstructing the terminus string can be done similar to local differencing, by iterating through the directives and constructing a temporary string [Tri99]. Remote synchronization can be done in-place similarly to local synchronization, requiring the read-write conflict resolving prior to beginning writing to the terminus string. Conversion of copy directives to insert directives will increase the size of delta encoding (communicated from version host to reference host). If the amount of conflicts is small, using temporary space allows a version host to transmit modified copy directives instructing to make a temporary copy of a conflicting chunk, to avoid overwriting the chunk prior to referring to it. Temporary space also allows the reference host to do the read-write conflict resolving, instead of version host. The temporary space is analoguous to the increase in delta encoding output size with local differential compression.

Reference host should transmit some initial notification message to version host to allow version host to begin fingerprinting its string concurrently with reference host, instead of waiting for multi-gibibyte reference string to complete fingerprinting [Tri99]. Without a notification, version host has no knowledge of the reference host. The message can also contain information required for correct operation of a differencing algorithm, such as parameters of chunk division function. No response is required, so the initial communication can be viewed as a prefix for the first main message, and does not incur the round-trip latency [Tri99]. Often practical implementations send a directory listing of files, and ignore synchronizing files with identical modification times [Tri99]. Single-round algorithms can begin sending the fingerprints while still fingerprinting. However, multi-round recursive comparison requires lower level fingerprinting to be completed before it can fingerprint the higher level. Characteristic polynomial interpolation requires full completion of the fingerprinting to be able to compute the characteristic polynomial, and thus benefits from a separate start notification. Similarly, reconstructing the terminus string can be started while receiving fingerprints, unless in-place reconstruction is required. In-place reconstruction is not possible until differencing is completed because read-write conflict resolving requires knowing the complete set of copy directives. When synchronizing multiple strings the latency can be masked by pipelining multiple synchronizations into one communication round, eliminating latency from connecting and protocol handshaking [TrM96]. Pipelining can improve synchronization speed by a factor of 100 with short strings [Tri99].

A synchronization algorithm should transmit a strong checksum of the whole string for verifying that reconstructed terminus string is identical to version string [Tri99]. The reference host will compute a corresponding checksum of the terminus string and compared it against the received checksum. When the checksum does not match, if a rolling hash function is used, it is re-initialized with other randomized polynomial [Tri99]. With cryptographic hash functions, fingerint size can be increased if less than full hash was used. Synchronization is then attempted again against the corrupted terminus string, instead of the original reference string [Tri99]. The corrupted terminus string is assumed to have only few differing chunks due to fingerprint collisions, less than the original reference string had prior to failed synchronization [Tri99, RaB03]. If (in-place) synchronization fails, the corrupted terminus string can be preserved as a renamed file and synchronization can later be retried against it, until succeeds [RaB03]. However, if cryptographic hash functions are used for fingerprinting and the failure was not a result of a transmission error, the failure was likely caused by a hash collision. In such case, the strings cannot be synchronized.

4.7 Optimizations

Random values have no redundancy between them. Checksums are thus incompressible by itself and no known encoding can be used to make the checksums more compact. Because fingerprints are assumed to be pseudo-random, they are not expected to be compressible and there is no need to entropy-encode fingerprints [YIS08]. However, the fingerprint values can be sorted into ordered list. Instead of sending the fingerprints themselves, the difference to preceding fingerprint value is sent using *Golomb coding* [YIS08]. For example, when an individual fingerprint needs 64 bits, if there are 2^{20} distinct fingerprints to be sent, the required size for successive fingerprints is expected to be ~44 bits per fingerint. Similar optimization can be used for chunk length and position information. Chunk positions can be given as relative offsets to preceding chunks. Assuming that chunk length variance is small, it is more efficient to transmit the difference to average chunk length (smaller number) instead of chunk length itself (larger number). Additionally, the sequences of integers can be compressed using some secondary compression algorithm.

For transmitting less fingerprint data with comparison-based differencing algorithms, Suel, Noel and Trendafilov [SNT04] suggest that the reference host can send only partial (weakened) fingerprints. Multiple successful comparisons can be then confirmed by calculating a strong checksum from the matched chunks using another communication round. A differing weakened fingerprint cannot be made identical by complementing with the missing part. The weakened matches are strong enough to identify candidate chunks for later verification, but not strong enough to verify by itself. Version host can then request chunk contents for the partial fingerprints that did not match and transmit the verification checksum on the same communication round. When the verification checksum does not match, the set of potentially matching chunks can be divided into smaller subsets (possibly of size 1), and verification checksums are calculated and transmitted again on the next round. Finding out which blocks had fingerprint collisions is closely related to the problem of group testing and binary search with errors [SNT04]. Suel, Noel and Trendafilov also note that verifying the weak matches on each communication round is not optimal, because temporary results help inferring which chunks may differ with higher probability (candidate matches). When a host receives weak hashes, it calculates respective weak fingerprints and responds to the other host with bitmaps of groups of candidate matches and strong verification checksums for the groups. The other host then responds with a bitmap of successfully matched verification checksums.

A differing chunk rarely exactly contains the differences: often there is slack on both the beginning and the end of a chunk. A neighboring matching chunk could be extended to remove some of the slack. Suel, Noel and Trendafilov [SNT04] suggest using weakened *continuation hashes* that allow extending matches with fewer bits than separate independent fingerprints. A *decomposable hash* [SBB90] function allows calculating hash values such that when having a chunk divided into two, a fingerprint for one sub-chunk can be calculated from the fingerprints of the other sub-chunk and the parent chunk: $h(right) = h(h(parent), h(left))$. Suel, Noel and Trendafilov use a modified Adler-32 checksum where bits that can be calculated from parent and sibling are masked out from being transmitted. This enables transmitting only one child fingerprint on each communication round and enables differencing with smaller chunk size that would require too much fingerprint data using stronger fingerprints. When there are multiple contiguous matching chunks, Suel, Noel and Trendafilov suggest that fewer than $\log_2 n$ bits are needed for a fingerprint because modifications are assumed spatially close. An algorithm should expand matching intervals incrementally, instead of expanding aggressively and attempting to filter out invalid candidates. They also note that processor computation speed may become the bottleneck, and thus even though less data is transmitted, synchronization completes slower.

Another approach called *half-block alignment* was presented by Irmak, Mihaylov and Suel [IMS05]. The idea is that when a non-matching chunk is found next to a matching chunk, it might be possible to extend the match halfway into the non-matching chunk, even though the complete chunk does not match. Also, when there are two non-matching chunks, there is possibility to find a match of two halves beginning from the latter half of the first non-matching chunk. Before reconciling fingerprints, the string is divided into half-length chunks and the fingerprint size is also divided in half. The remaining halves are not requested when the matching other halves of fingerprints belong to contiguous sequences of matches. With half-size fingerprints, one match is not enough to provide confidence because the fingerprint is too weak. The algorithm considers at least two consecutive matches as sufficiently reliable. Irmak, Mihaylov and Suel also suggest that other sub-divisions are possible, but not likely to result in significant improvement in compression.

When single-round communication is required with comparison-based fingerprint differencing, it is possible to use fewer bits for communication by emulating multi-round communication. Using only a single communication round, a differencing algorithm sends additional information that can be used to reconstruct missing data up to some pre-chosen threshold amount. This is based on the assumption that there are only a maximum of d differing chunks. Irmak, Mihaylov and Suel [IMS05] presented a method of using *erasure codes* to encode error recovery information allowing recovering a maximum of d intentionally discarded fingerprint values by sending additional bits. The idea is similar to error correction, where k extra fingerprints are calculated to allow recovering up to d missing fingerprints, except that now the fingerprints are intentionally excluded. An initial message (set of fingerprints) from other host can be used to estimate d and to choose k. A host partitions its string using fixed-width chunks as it normally would for recursive differencing and calculates the fingerprints. The version host then calculates $2k$ additional erasure fingerprints per recursion level, except for the root level. The version host then transmits the root level fingerprints and the erasure fingerprints. Using $k + x < n$ bits allows recovering $y < x$ bits of missing fingerprints.

The algorithm of Irmak, Mihaylov and Suel [IMS05] emulates a recursive algorithm that sends all of the fingerprints, not just those whose parent fingerprint did not match. The algorithm thus requires sending more data than true recursive comparison, but less than single-round algorithms. During differencing the other host calculates descendant fingerprints for the matching chunks and erasure fingerprints are used to recover any of the missing fingerprints when recursion progresses deeper. If there are more than k differing parent chunks, recursion cannot continue and the algorithm stops recursing and requests content for the remaining chunks, regardless of on what level the recursion was. When recursion is stopped the chunk size can remain large, but the algorithm is still able to complete differencing.. This algorithm is communication efficient as defined earlier and requires sending fewer bits than e.g., rsync. An emulated algorithm however cannot match the compression ratio of true multi-round differencing. The emulation has an advantage over true multi-round: erasure codes allow multicasting, where the same communication content can be used to synchronize multiple different reference strings, because the k missing chunks need not always be the same ones.

5 Empirical Comparison

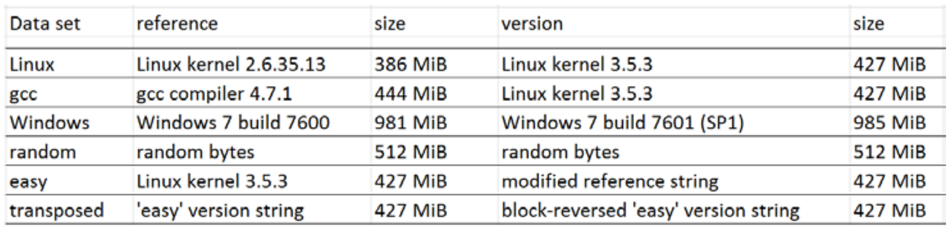
Typically asymptotic bounds, including those given in this thesis, are based on evaluating the algorithms with random input strings. Average case analysis based on random content is problematic, because practical inputs are not random and contain repetitive patterns [YIS08]. Implementation specifics can also have a large effect on execution time, memory requirement and compression ratio [BSL02, You06]. Thus, we perform empirical evaluation of differential compression algorithms and present the results.

We implemented a subset of algorithms discussed in this thesis to test how operating parameters affect results. We tested our implementation and a number of existing popular implementations on various input samples to find out how the algorithms perform on current hardware. In section 1 we describe the specifications for the empirical comparison: the test data sets we chose, the algorithm implementations and their features, and the environment we ran the tests. Finally, in section 2 we present the measurement results and give our conclusions on the results.

5.1 Test Specifications

We chose six different data sets for testing (see figure 31). For each data set, we concatenated all of the files into a single file by iterating through each directory and then each file in lexicographical order. Each data set is thus a pair of strings. First, we chose two large Linux kernel source code distributions to test with transposed repeats, extensive modifications and large number of intra-repeats. The second data set *gcc* has two strings of similar type, but not from the same software: we used gcc as the reference string and Linux kernel source as the version string. The strings have many short repeats, such as C/C++ language function declarations, but not many longer repeats as they don’t share the same codebase. The third data set simulates a binary software update (patch) and consists of all executable and dynamic link library files from ‘system32’ directory of Windows 7 Pro 64-bit (English) operating system as reference string, and the respective set of updated files from Service Pack 1 version of the same operating system. As a reference, the Service Pack 1 distribution download package from Microsoft is 903 MiB in size.

Additionally, we included three synthetic tests with artificial strings. For the fourth data set we generated two distinct pseudo-random strings of 512 MiB each. We used OpenSSL to generate the strings. For the fifth ‘easy’ data set we used the Linux 3.5.3 source code as reference string, and created a versions string by modifying the reference string by performing a search and replace to change all substrings “unsigned” into “different”. The modification replaced 258 958 substrings, fragmenting the version string into short repeats separated by the same intra-repeat. As the substring “different” already exists in the reference string, the optimal differencing result would thus be sequence of copy directives, where each directive could use short relative offsets for position information. For the sixth data set we created transpositions by first dividing the ‘easy’ version string into chunks and then concatenating the chunks in reverse order. Each chunk starts with the word ‘different’ (except the first one). The version string thus does not contain any modifications except for transpositions, and an optimal differencing algorithm should match the whole version string as repeats and produce a sequence of copy directives.



Data set	reference	size	version	size
Linux	Linux kernel 2.6.35.13	386 MiB	Linux kernel 3.5.3	427 MiB
gcc	gcc compiler 4.7.1	444 MiB	Linux kernel 3.5.3	427 MiB
Windows	Windows 7 build 7600	981 MiB	Windows 7 build 7601 (SP1)	985 MiB
random	random bytes	512 MiB	random bytes	512 MiB
easy	Linux kernel 3.5.3	427 MiB	modified reference string	427 MiB
transposed	'easy' version string	427 MiB	block-reversed 'easy' version string	427 MiB

Figure 31: Six data sets for evaluating differential compression algorithms' performance.

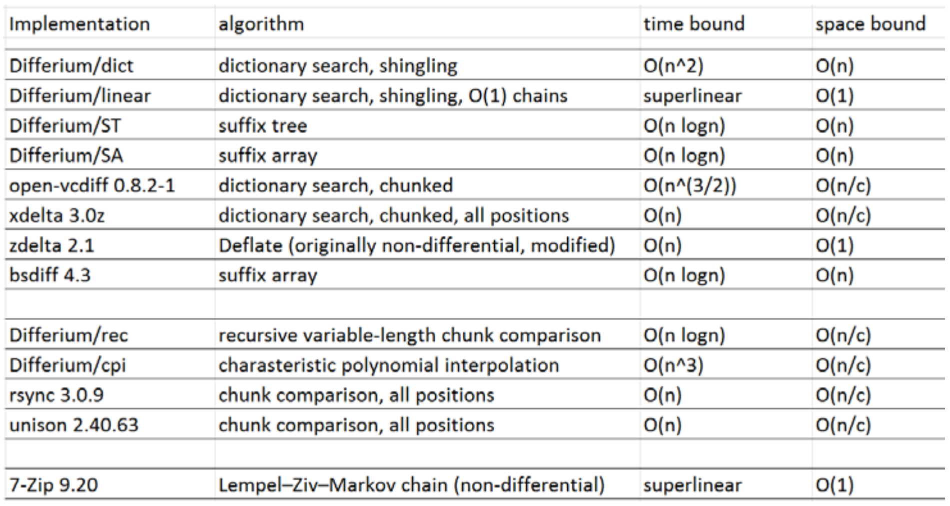
The differential compression implementations we test cover the main differencing algorithms discussed in this thesis (see figure 32). For all implementations, we verified that the delta encoding can be used to reconstruct the terminus string. We used our implementation, Differium, to test algorithms that we could not find a public implementation. The first algorithm we implemented is Obst's greedy algorithm using dictionary searching (Differium/dict). Using 12 byte seed prefixes, we are able to optimally find all repeats longer than 11 bytes. We chose a dictionary size of 32459641 entries (a prime number) and used xdelta's fingerprinting function, as we tested it was faster than Rabin and produced fewer collisions. For the next algorithm (Differium/linear) we then limited the dictionary chain length to 10 entries, using a collision policy that removes the oldest entry when the chain length limit is reached. Our third algorithm (Differium/ST) uses a suffix tree to optimally find all repeats longer than four bytes from the strings. We also implemented a suffix array algorithm, denoted as Differium/SA, which is similar to Differium/ST except for the array in place of a suffix tree. We implemented a simple delta encoding method using VCDiff integer encoding described in section 3.5. Copy directive positions are encoded relative to previous copy directive, and we do not attempt to optimize by choosing from multiple possible repeats. Due to inefficiency of the encoding method, we do not attempt to reference shorter than four-byte repeats. For secondary compression, we used 7-Zip to compress using LZMA2 algorithm.

Additionally, we included four other implementations. Open-vcdiff's[5] algorithm [BeM99] is based on dictionary searching, but instead of searching for short seed prefixes, it divides a string into fixed-length non-overlapping chunks and uses the dictionary for the chunk fingerprints. The resulting delta output is compressed using external third-party non-differential compressor [AGJ05]. The bsdiff[6] algorithm is based on suffix array searching [Per03]. It uses Larsson and Sadakane's suffix sorting algorithm to construct two suffix arrays (one for each string). The bsdiff algorithm requires $\mathcal{O}(n)$ space and $\mathcal{O}((n+m)\log n)$ time [Per03]. The xdelta[7] algorithm is based on a chunk comparison remote differencing algorithm, but applies it for local differencing using short chunk length [Tri99, AGJ05]. The algorithm is designed to

[5] http://code.google.com/p/open-vcdiff/
[6] http://www.daemonology.net/bsdiff/
[7] http://xdelta.org

prioritize speed over compression ratio and has $O(n)$ time and space bound [Mac99]. The zdelta[8] algorithm is modified from a non-differential compressor (zlib) and uses the *deflate* algorithm with a “process but do not emit” instruction [AGJ05]. Zdelta performs entropy encoding using Huffman. The algorithm has $O(n)$ time and $O(1)$ space bounds [AGJ05].

For remote synchronization, we chose two implementations to compare against, in addition to implementing two ourselves. Both Rsync[9] and Unison[10] use a similar differencing algorithm. The algorithm divides the strings into fixed-length chunks and compares the chunks from version string against every position of reference string, using two-level fingerprinting. The algorithm has a $O(n)$ time bound and $O(n/c)$ space bound (where c is the chunk size). Our recursive fingerprint comparison implementation (Differium/rec) uses content-defined local-maxima chunk division function. We chose a horizon length of 250 based on empirical evaluation of average chunk sizes using various horizon lengths. The last of our implementations is based on interpolating the characteristic polynomial (Differium/cpi), and uses the algorithm described by Yan, Irmak and Suel [YIS08]. We used the NTL 5.5.2 number theory library for factoring the results. In addition to these differential compressors, we chose a non-differential compressor as a reference benchmark. 7-Zip 9.20 is used to compress the version string only, representing a use case where a user does not use differential compression.



Implementation	algorithm	time bound	space bound
Differium/dict	dictionary search, shingling	O(n^2)	O(n)
Differium/linear	dictionary search, shingling, O(1) chains	superlinear	O(1)
Differium/ST	suffix tree	O(n logn)	O(n)
Differium/SA	suffix array	O(n logn)	O(n)
open-vcdiff 0.8.2-1	dictionary search, chunked	O(n^(3/2))	O(n/c)
xdelta 3.0z	dictionary search, chunked, all positions	O(n)	O(n/c)
zdelta 2.1	Deflate (originally non-differential, modified)	O(n)	O(1)
bsdiff 4.3	suffix array	O(n logn)	O(n)
Differium/rec	recursive variable-length chunk comparison	O(n logn)	O(n/c)
Differium/cpi	charasteristic polynomial interpolation	O(n^3)	O(n/c)
rsync 3.0.9	chunk comparison, all positions	O(n)	O(n/c)
unison 2.40.63	chunk comparison, all positions	O(n)	O(n/c)
7-Zip 9.20	Lempel–Ziv–Markov chain (non-differential)	superlinear	O(1)

Figure 32: Summary of algorithm implementations included in empirical comparison.

Tests were run under Windows Server 2008 R2 SP1 operating system, except for zdelta and open-vcdiff that were run under Debian “wheezy” Linux. The compression tests were performed on a machine equipped with Intel 2600K processor running at 5.0 GHz, having 32 GiB of RAM and an OCZ Vertex3 MaxIOPS 128GiB solid-state drive. For remote differential

[8] cis.poly.edu/zdelta/
[9] http://rsync.samba.org/
[10] http://www.cis.upenn.edu/~bcpierce/unison/

compression, the same machine was assigned both reference host and version host roles. For testing, power-saving features were disabled, forcing the processor cores to run at maximum clock frequency. We stopped all non-essential daemons and waited for one minute before starting each test. For each test, the machine was rebooted prior to testing to clear the processor and operating system caches. Three attempts were done for each test if the running time was less than fives minutes and the fastest result was chosen (any increase in execution time would be due to some external process). We implemented our algorithms using C++ language and Visual C++ 2010 SP1 compiler. The compilation was done using “maximum speed” optimization option.

5.2 Measurement Results

We measured the size of delta encodings produces by the differential compressor implementations (see figures 33 and 34). When a compressor had an option to produce delta output with secondary compression algorithm, we enabled the option. We present results without secondary compression separately. The ‘easy’ test was the only one where local differential compressors could produce smaller compression ratio than the 7-Zip compressor, even though 7-Zip does not have the advantage of having a reference string available. There are also big differences in results produced by similar algorithms. For example, both Unison and Rsync are based on the same algorithm principle, but Unison’s results are worse by a factor of ~4. Similarly, xdelta and open-vcdiff are both based on dictionary search of chunks, but they produce results that differ up to a factor of ~40. We thus infer that implementation details are important in practice. On average, the most efficient local compressor was bsdiff, which is based on a suffix array algorithm. Using bsdiff, the Windows 7 Service Pack 1 could be stored using ~97% less space than it currently uses. Of the remote compressors, the most efficient was Differium algorithm based on recursive fingerprint comparison. However, the advantage was due to Differium using LZMA2 algorithm for secondary compression, not because Differium found fewer differences. Even though remote differencing includes slack, the results from Differium/rec are not much worse compared to local differential compressors, except for the ‘easy’ data set. This again is the result of using efficient secondary compression (see figure 37). For reference, we measured the number of differing symbols by finding all repeats longer than 3 bytes using Differium/SA algorithm (given as optimal/4 in the table). Optimal/4 results do not include any encoding directives and are not secondary compressed.

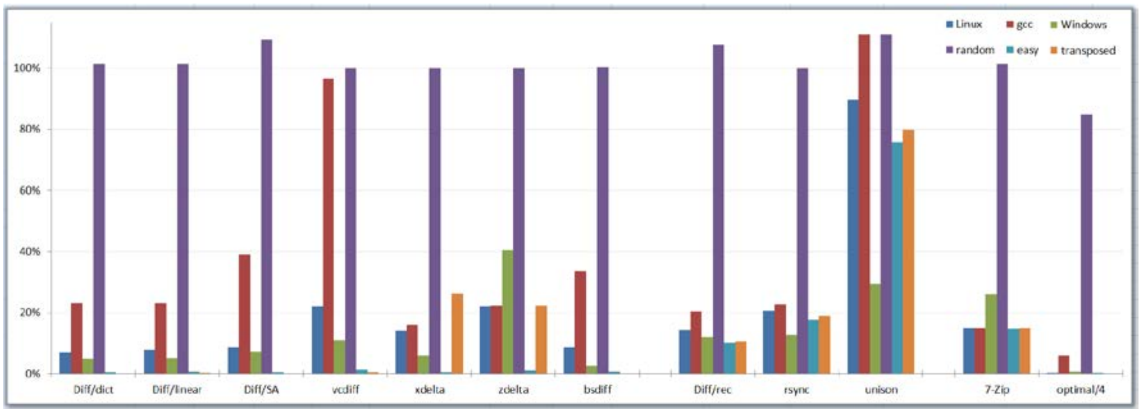
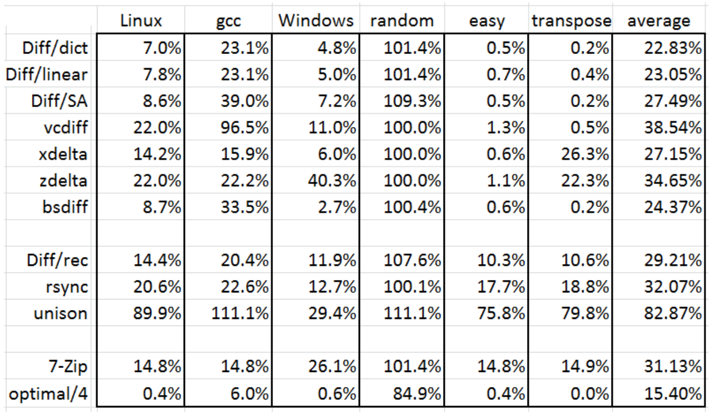


Figure 33: Compression ratios of delta encodings produced by the compressor implementations.



	Linux	gcc	Windows	random	easy	transpose	average
Diff/dict	7.0%	23.1%	4.8%	101.4%	0.5%	0.2%	22.83%
Diff/linear	7.8%	23.1%	5.0%	101.4%	0.7%	0.4%	23.05%
Diff/SA	8.6%	39.0%	7.2%	109.3%	0.5%	0.2%	27.49%
vcdiff	22.0%	96.5%	11.0%	100.0%	1.3%	0.5%	38.54%
xdelta	14.2%	15.9%	6.0%	100.0%	0.6%	26.3%	27.15%
zdelta	22.0%	22.2%	40.3%	100.0%	1.1%	22.3%	34.65%
bsdiff	8.7%	33.5%	2.7%	100.4%	0.6%	0.2%	24.37%
Diff/rec	14.4%	20.4%	11.9%	107.6%	10.3%	10.6%	29.21%
rsync	20.6%	22.6%	12.7%	100.1%	17.7%	18.8%	32.07%
unison	89.9%	111.1%	29.4%	111.1%	75.8%	79.8%	82.87%
7-Zip	14.8%	14.8%	26.1%	101.4%	14.8%	14.9%	31.13%
optimal/4	0.4%	6.0%	0.6%	84.9%	0.4%	0.0%	15.40%

Figure 34: Compression ratios of delta encodings produced by the compressor implementations.

We failed to get results from our Differium/cpi implementation. Charasteristic polynomial interpolation requires too much computation with our test data sets, as the number of differences is too large for an algoritim requiring cubic execution time. We stopped testing after waiting 10 hours. We tested the implementation separately with data specifically fabricated for the CPI algorithm. Differencing between two sets of 10^6 fingerprints with 500 differences was executed in 47 seconds and required 4.6 kibibytes of data from reference host to version host, after which version host had the required information to send the chunk substrings. Recursive fingerprint comparison would require 195 kB for fingerprints using three communication rounds when assuming even distribution of differences, 128-bit fingerprints and 130 bytes per fingerprint-level chunk (we do not account chunk contents at all). Charasteristic polynomial interpolation would be superior in cases where number of differences is known to be low.

Differium/linear produced larger delta encodings due to not being able to match as many repeats due to limited chain length, but secondary compression almost eliminated the difference. Differium/ST produced identical encoding results to Differium/SA, as both are able to find repeats of any length. However, the suffix array version fast faster and required less memory. Directives for short copys found by Differium/SA did not compress as well as fewer longer copy directives by Differium/dict, which resulted in worse overall results for Differium/SA, even though the uncompressed delta encoding was shorter. To see how many repats are missed due to Differium/dict able to find only repeats longer than 11 bytes, we used our optimal suffix array searching algorithm to count the number of repeats by their lengths (see figure 35). In the Linux test there are 5484090 repeats shorter than 12 symbols, consuming ~36.4 mebibytes or ~8.5% of the string.

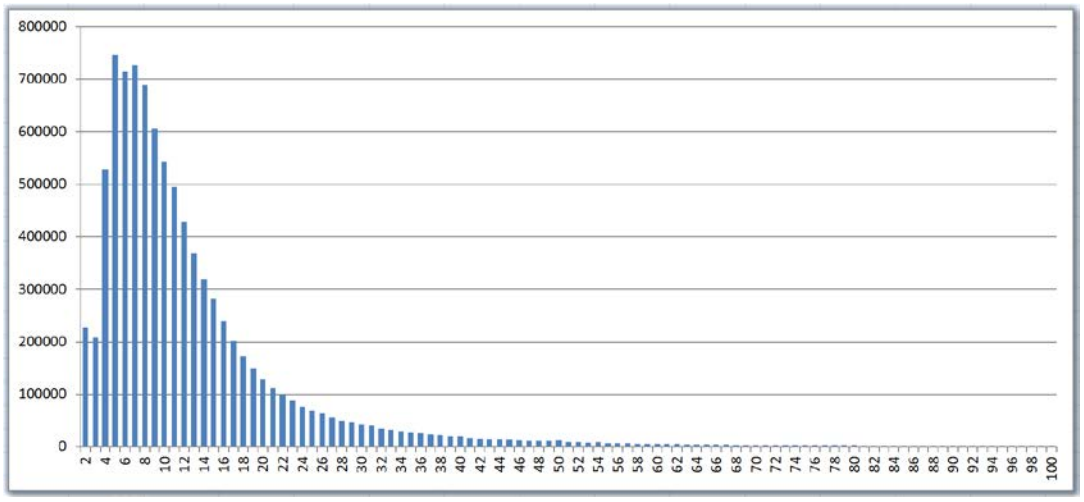
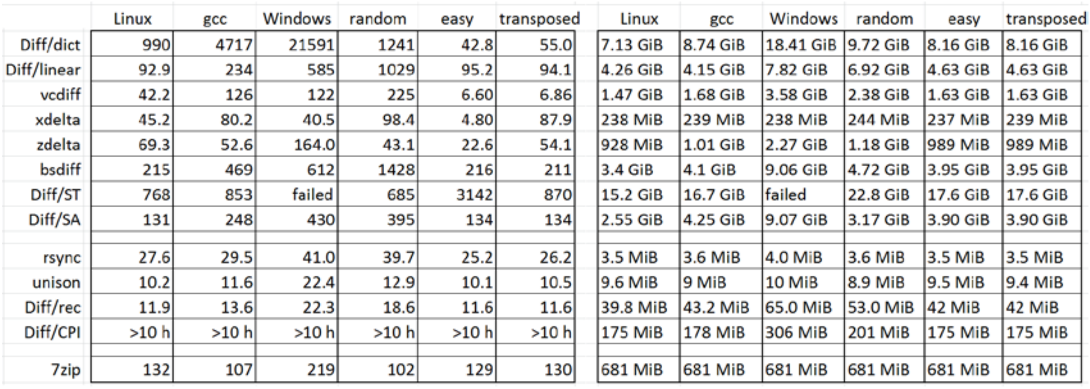


Figure 35: Number of repeats by length from Linux dataset (up to length of 100 symbols).

We measured the peak memory consumption and compression time for each compressor and test case (see figure 36). Of the public implementations, bsdiff produced the smallest compression ratio on average, but also requires a lot of memory. Note that our Differium implementations were made specifically for evaluating the algorithms and memory consumption was not optimized.



	Linux	gcc	Windows	random	easy	transposed	Linux	gcc	Windows	random	easy	transposed
Diff/dict	990	4717	21591	1241	42.8	55.0	7.13 GiB	8.74 GiB	18.41 GiB	9.72 GiB	8.16 GiB	8.16 GiB
Diff/linear	92.9	234	585	1029	95.2	94.1	4.26 GiB	4.15 GiB	7.82 GiB	6.92 GiB	4.63 GiB	4.63 GiB
vcdiff	42.2	126	122	225	6.60	6.86	1.47 GiB	1.68 GiB	3.58 GiB	2.38 GiB	1.63 GiB	1.63 GiB
xdelta	45.2	80.2	40.5	98.4	4.80	87.9	238 MiB	239 MiB	238 MiB	244 MiB	237 MiB	239 MiB
zdelta	69.3	52.6	164.0	43.1	22.6	54.1	928 MiB	1.01 GiB	2.27 GiB	1.18 GiB	989 MiB	989 MiB
bsdiff	215	469	612	1428	216	211	3.4 GiB	4.1 GiB	9.06 GiB	4.72 GiB	3.95 GiB	3.95 GiB
Diff/ST	768	853	failed	685	3142	870	15.2 GiB	16.7 GiB	failed	22.8 GiB	17.6 GiB	17.6 GiB
Diff/SA	131	248	430	395	134	134	2.55 GiB	4.25 GiB	9.07 GiB	3.17 GiB	3.90 GiB	3.90 GiB
rsync	27.6	29.5	41.0	39.7	25.2	26.2	3.5 MiB	3.6 MiB	4.0 MiB	3.6 MiB	3.5 MiB	3.5 MiB
unison	10.2	11.6	22.4	12.9	10.1	10.5	9.6 MiB	9 MiB	10 MiB	8.9 MiB	9.5 MiB	9.4 MiB
Diff/rec	11.9	13.6	22.3	18.6	11.6	11.6	39.8 MiB	43.2 MiB	65.0 MiB	53.0 MiB	42 MiB	42 MiB
Diff/CPI	>10 h	>10 h	>10 h	>10 h	>10 h	>10 h	175 MiB	178 MiB	306 MiB	201 MiB	175 MiB	175 MiB
7zip	132	107	219	102	129	130	681 MiB	681 MiB	681 MiB	681 MiB	681 MiB	681 MiB

Figure 36: Execution time (in seconds) needed to compress a version string and generate a delta encoding, with memory requirements, of differential compressor implementations.

Some of the compressors had an option to disable secondary compression. We measured how much secondary compression lowers the compression ratio (see figure 37). We measured both uncompressed delta encoding size, and additionally compressed the uncompressed delta encodings with 7-Zip to see if any further size reduction was possible. Based on the results, we infer that secondary compression is essential to remote differencing.

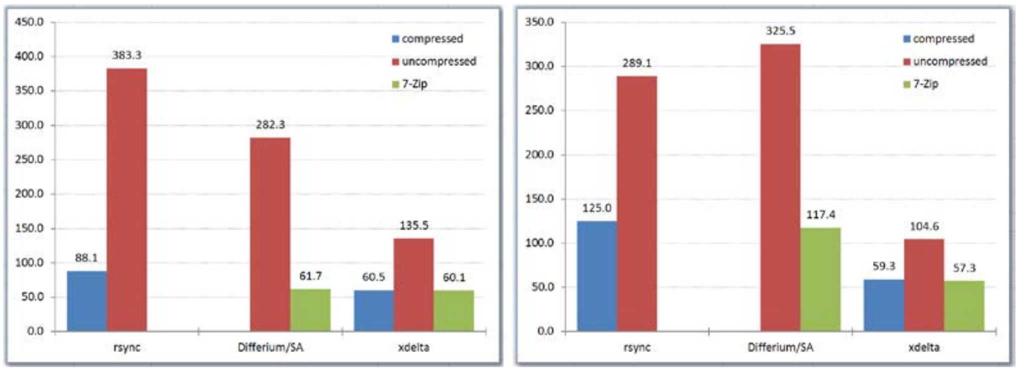


Figure 37: Compression ratios of delta encodings after secondary compression. The left chart is for Linux dataset, and the right chart is for Windows dataset.

We tested various horizon sizes for local-maxima chunk division function and measured both average chunk size and amount of slack for each block size (w); see figure 38. Slack decreases as block size decreases, but protocol overhead increases due to increasing number of fingerprints. Block size of 125 lead to smallest total communication requirement for Linux dataset, but the size is content-dependent. We also measured slack for all datasets (see figure 39). Our Differium/rec implementation failed to find most repeats due to long atomic unit of change.

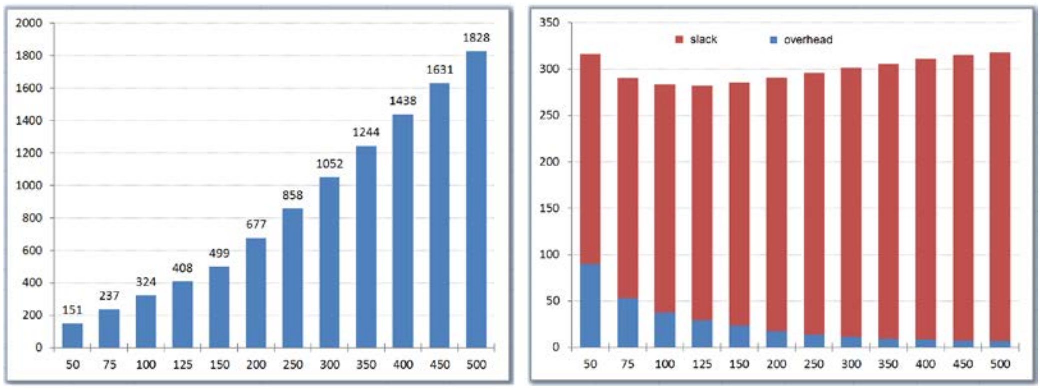


Figure 38: Average chunk size in bytes (left), and delta encoding sizes (in MiB) produced by chunk division (right), measured for range of local-maxima chunking block lengths.

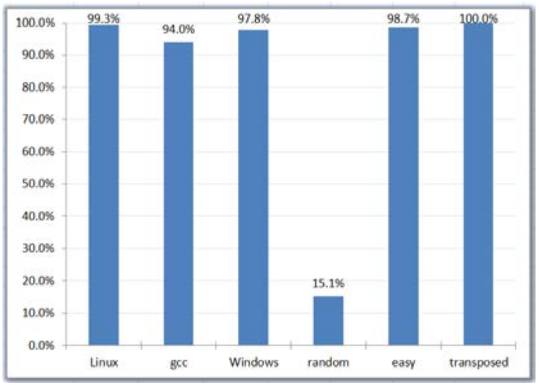
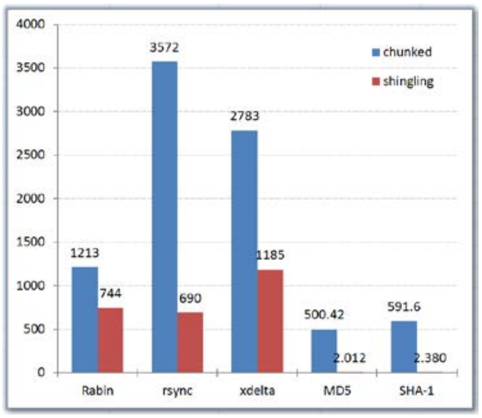
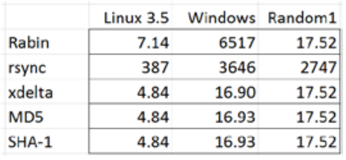


Figure 39: Slack for tested datasets from Differium/rec implementation.

Hash function’s speed and uniformity of hashes is relevant for both local and remote differencing. To test uniformity of hashes, we measured the variance of dictionary chain lengths with a modified Differium/dict algorithm. The modification allows inserting fingerprints only from unique seed prefixes, thus eliminating all hash collisions due to identical seed prefixes. We measured commonly used hash functions (see figure 40). Rsync weak hash function causes significantly more hash collisions compared to others. Additionally, we tested how much the two-level fingerprinting scheme improves synchronization speed on modern hardware. We used a SHA-1 implementation provided by Intel to compute 160 bit strong fingerprints and Rabin to for 32 bit weak fingerprints. The SHA-1 was faster than MD5 due to being optimized for given hardware. Chunked hashing was done by computing hashes of 256 byte chunks, initializing the hash function for each chunk. Shingling hashing was performed by initializing the hash function on every byte position, and computing (overlapping) hashes of 256 byte chunks. Our measurements indicate the two-level fingerprinting scheme used in rsync is still advantageous if network bandwidth is above ~40 kiB per processor core, per second. One core can compute ~9700 hashes per second, corresponding to 38.8 kiB of weak fingerprints per second.





	Linux 3.5	Windows	Random1
Rabin	7.14	6517	17.52
rsync	387	3646	2747
xdelta	4.84	16.90	17.52
MD5	4.84	16.93	17.52
SHA-1	4.84	16.93	17.52

Figure 40: Throughput of hash functions using single execution thread (left). Variance of chain lengths (right).

6 Conclusion

The main challenges in differential compression are long strings, recurring patterns with uneven distribution of symbols, granularity of changes, and transpositions. Information collected from long strings cannot fit into memory, and discarding information makes it impossible to find all repeats. Dense changes prevent using longer atomic unit of change, which in turn prevents saving memory by summarizing information using fingerprints. A transposition may relocate a substring far away from original location, making matching it problematic when information from complete string cannot be kept in memory. Solving the differencing problem is about developing heuristics that simulate optimal algorithms as closely as possible, while using the least amount of memory possible. Differential compression requires compromising and choosing between multiple trade-offs. There is a three-way trade-off between compression ratio, execution time and memory requirement.

There are two local differencing algorithms capable of perfect differencing by finding all repeats between two strings, and they both have either time or space asymptotic bound that prevents using them with arbitrarily long strings. There is no known optimal differencing algorithm for remote differencing due to long atomic unit of change. It is possible to solve the local differencing problem optimally either in $\mathcal{O}(n^2)$ time and $\mathcal{O}(1)$ space, or in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space. The greedy algorithm that compares all strings positions against all positions of another string is too slow because of $\mathcal{O}(n^2)$ time bound. With remote differencing, shortening the atomic unit of change would be possible with a nearly communication-optimal algorithm, but the charasteristic polynomial interpolation algorithm requires $\mathcal{O}(n^4)$ time in the general case. The algorithm also has a linear memory requirement, making it unsuitable for arbitrary long strings. For specialized use cases the algorithm can be the best choice, as its communication requirement is not dependent on string lengths.

The optimal greedy local algoritm can be approximated with a dictionary search algorithm that keeps repeat positions in the dictionary and finds the positions using a summary function. Information can be limited by discarding old information, refusing to collect new information, or collecting information from sparse intervals. Multiple heuristics have been developed for deciding which fingerprints to discard from dictionary. A sliding window approach commonly used with non-differential compression can be used to limit memory requirements. Suffix tree algorithms can be substituted with suffix array algorithms, which require much less memory and are faster in practice. Suffix arrays are still large and have linear asymptotic space bound, but in our empirical testing we got the smallest compression ratio from suffix array algorithms. Compared to dictionary searching, suffix array creation algorithms are complex and thus slower. Dictionary searching can find more repeats in low memory situations. There are compressed suffix arrays, which require less space than the original string, and which are currently researched heavily. However, creating the arrays is computationally expensive. There is also an interesting algorithm based on projecting string as discrete binary signals and using cyclic correlation to detect possible repeats. The algorithm is however not competitive with either dictionary or suffix searching as it requires more memory than dictionaries and is slower than suffix searching.

Remote differencing is probabilistic, and there is always a small probability of undetected corrupted results. For remote differencing, the best algorithm in practice is recursive fingerprint comparison of chunks. Dividing the strings into chunks using local-maxima content-defined division function makes the chunks resistant to alignment problems and does not require comparing chunks from one string against every position of another string. Recursion allows extending known long repeats into neighboring chunks and optimizing fingerprint sizes by initially using weaker fingerprints, as matches can be verified later. Due to round-trip latency, recursion should not be used with short strings, as several kibibytes could be transferred during the delay. Charasteristic polynomial interpolation would be superior to fingerprint comparison algorithms when the number of differences is low, but the problem is that the number is not known beforehand. Estimating the number of differences requires transmitting sample fingerprints, which reduces communication efficiency. Estimation is also not reliable, and under-estimating causes the algorithm to fail, while over-estimating causes a lot of unnecessary computation.

Choosing the most suitable algorithm for differencing and encoding is dependent on several variables, which implies that one algorithm cannot be the best choice for all cases. Variables should be evaluated empirically and algorithms and heuristics chosen at runtime. We also propose that multiple differencing algorithms could be combined into a hybrid implementation. For example, an algorithm could optimistically use sparse dictionary searching to find long repeats, and then construct a suffix array of the remaining non-matched substrings. This, potentially much shorter array could then be matched against the other string with dense granularity. Using just suffix tree would not be possible due to memory requirement, and using just dictionary searching would not find all repeats. For practical implementations, the implementation details are important. An implementation of the same algorithm can produce compression ratios that differ by a factor of 10. Avoiding random accessing the strings is important, as it can slow down an algorithm by a factor of 100. Avoiding unnecessary copying of unchanged data and not requiring large temporary space can be avoided by using in-place reconstruction with read-write conflict-free delta encoding.

Considering that the problem of differential compression is close to conventional compression, combining the compressors would likely produce a superior differential compressor. The hybrid compressor could operate as both differential and non-differential compressor, and use the same efficient non-differential algorithms for delta encoding. Compressing a completed delta encoding is not as efficient as using a specialized compression algorithm, as the compression predictor model can access the surrounding string context while differencing. The surrounding string is no longer available in the finished delta encoding. In our empirical testing, a non-differential compressor (7-Zip) produced smaller outputs in several cases, even though it did not access the reference string at all. Some of the heuristics discussed in this thesis are almost 10 years old, but we could not find any applications that implement them.

Even though optimal algorithms are known, there remain unsolved problems. It is not known whether an optimal chunk division algorithm with respect to slack exists [TBG06]. Methods for remotely estimating number of differences could be improved, to allow general use of characteristic polynomial interpolation. It also has not been shown that an optimal differencing algorithm with linear time bound and constant space bound could not be created [ABF02]. The heuristics can also be improved. New succinct suffix array construction algorithms may allow using suffix searching in more general use scenarios in the future. We conclude by suggesting that current implementations could be much improved by implementing heuristics discussed in this thesis and by using more efficient secondary compression, as currently non-differential compressors often produce better results.

List of References

AKO04	Abouelhodaa, M. I., Kurtz, S., & Ohlebusch, E. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, Volume 2, Issue 1, March 2004, pages 53–86.
Ada09	Adams, S. Chromium Blog: Smaller is Faster (and Safer Too). URL: http://www.chromium.org/developers/design-documents/software-updates-courgette, [2009, July 15]
Ste09	Adams, S. Chromium Projects: Software Updates: Courgette. URL: http://dev.chromium.org/developers/design-documents/software-updates-courgette, [2009]
AGJ05	Agarwal R.C., Gupta K., Jain S., & Amalapurapu S. An approximation to the greedy algorithm for differential compression. *IBM Journal of Research and Development*, Volume 50, Issue 1, January 2005, pages 149-166.
AAJ04	Agarwal, R. C., Amalapurapu, S., & Jain, S. An Approximation to the Greedy Algorithm for Differential Compression of Very Large Files. Technical report, IBM, 2004.
AgT06	Agarwal, S., & Trachtenberg, A. Approximating the number of differences between remote sets. *Proc. Information Theory Workshop,* March 2006, pages 217-221.
ACT06	Agarwal, S., Chauhan, V., & Trachtenberg, A. Bandwidth Efficient String Reconciliation using Puzzles. *IEEE Parallel and Distributed Systems,* November 2006, pages 1217-1225.
Bak93	Baker, B. S. On Finding Duplication in Strings and Software, Technical Report, 1993.
BMM99	Baker, B., Manber, U., & Muth, R. Compressing Differences of Executable Code, *ACM SIGPLAN Workshop on Compiler Support for System Software*, April 1999.
BaP98	Balasubramaniam, S., & Pierce, B. C. (1998). What is a File Synchronizer? *Proc. 4th ACM/IEEE International Conference on Mobile computing and networking MobiCom'98*, 1998, pages 98-108.
BeM99	Bentley, J., & Mcilroy, D. Data Compression Using Long Common Strings. *Proc. Data Compression Conference*, March 1999, pages 287-295.
BBG06	Bjorner N., Blass. A., Gurevich Y. Content-Dependent Chunking for Differential Compression, The Local Maximum Approach. Technical Report, Microsoft Research, 2006.
Bla06	Black, J. Compare-by-Hash: A Reasoned Analysis. *Proc USENIX Annual Technical Conference*, 2006, pages 85-90.
Bro93	Broder, A. Z. Some applications of Rabin's fingerprinting method. Proc. *Sequences II: Methods in Communications, Security, and Computer Science*, 1993, pages 143-153.
BuL97	Burns, L. A linear time, constant space differencing algorithm. *Proc. Performance, Computing, and Communications Conference (IPCCC)*, February 1997, pages 429-436.
BSL02	Burns, R., Stockmeyer, L., & Long, D. D. In-Place Reconstruction of Version Differences, *IEEE Knowledge and Data Engineering, July-August 2003*, pages 973-984.
BuW94	Burrows, M., & Wheeler, D. A block sorting lossless data compression algorithm, Technical Report SRC-RR-124, HP Labs, 1994.

ChM06 Chedid, F. B., & G.Mouawad, P. On Compactly Encoding With Differential Compression, *Proc. IEEE Computer Systems and Applications*, 2006, pages 123-129.

CLR01 Cormen, L. R. Introduction to Algorithms, 2nd edition. MIT Press, ISBN 0-262-03293-7, 2001, Section 16.3, pages 385–392.

CoJ05 Cox, R., & Josephson, W. File Synchronization with Vector Time Pairs, Technical Report MIT-CSAIL-TR-2005-014, February 2005.

ShS04 Shapira, D., & Storer, J.A. In-Place Differential File Compression of Non-Aligned Files With Applications to File Distribution, Backups, and String Similarity. *Proc. Data Compression Conference (DCC)*, March 2004, pages 82-91.

TBG06 Teodosiu, D., Bjørner, N., Gurevich, Y., Manasse, M., Porkka J. Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression, Technical Report MSR-TR-2006-157, Microsoft Research, 2006.

TMS02 Trendafilov, D., Memon, N., Suel, T., zdelta: An Efficient Delta Compression Tool. Technical Report TR-CIS-2002-02, June 2006.

EsT05 Eshghi, K., Tang, H. K. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Technical Report HPL-2005-30R1, HP Labs, 2005.

Fim02 Fimbel, E. Edit distance and Chaitin-Kolmogorov difference. Technical Report ETSRT-2002-001, 2002.

GaJ79 Garey, M., & Johnson, D. M.R. Computers and Intractibility: A Guide to the Theory of NP-Completeness, ISBN 0716710447, 1979.

GrV00 Grossi, R., & Vitter, J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *Proc. 32nd ACM Symposium on the Theory of Computing*, 2000, pages 397-406.

Hec78 Heckel, P. A technique for Isolating Differences Between Files. *Communications of the ACM*, Volume 21, Issue 4, April 1978, pages 264-268.

Hen03 Henson, V. An Analysis of Compare-by-hash. *Proc. 9th conference on Hot Topics in Operating Systems (HOTOS)*, Volume 9, July 2003, pages 13-18.

Hir75 Hirschberg, D. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, Volume 18, Issue 6, June 1975, pages 341-343.

HuM76 Hunt, J., McIlroy, M. An algorithm for Differential File Comparison. Technical Report, AT&T Bell Labs, 1976.

Int10 Intel Corporation. Improving the Performance of the Secure Hash Algorithm (SHA-1). URL: http://software.intel.com/en-us/articles/ improving-the-performance-of-the-secure-hash-algorithm-1/, March 2010.

KaR87 Karp, R., Rabin, M. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, March 1987, pages 249-260.

KoV02 Korn, D., & Vo, K.-P. Engineering a Differencing and Compression Data Format. *Proc. USENIX Annual Technical Conference*, 2002, pages 219-228.

KMM02 Korn D., MacDonald J., Mogul J. The VCDIFF Generic Differencing and Compression Data Format, RFC 3284.

Kur99 Kurtz, S. Reducing the space requirements of suffix trees. *Software – Practice and Experience*, Volume 29, Issue 13, November 1999, pages 1149–1171.

KAO04 Kurtz S., Abouelhoda M., & Ohlebusch E. Replacing Suffix Trees with Enhanced Suffix Arrays, *Journal of Discrete Algorithms*, Volume 2, Issue 1, March 2004, pages 53-86.

Lan01 Langford, J. Multiround Rsync. Unpublished paper, November 2001.

Lar96 Larsson, N. Extended Application of Suffix Trees to Data Compression. *Proc. Data Compression Conference*, March 1996, pages 190-199.

LKT06 Lindholm, T., Kangasharju, J., & Tarkoma, S. Fast and Simple XML Tree Differencing by Sequence Alignment. *Proc. ACM symposium on Document engineering*, 2006, pages 75-84.

Lin07 Linhard, M. (2007). Data structure for representation of maximal repeats in strings. Diploma Thesis, Comenius University, Bratislava, 2007.

Lot08 Lothar, M. (2008). Delta Compression of Executable Code Analysis, Implementation and Application-Specific Improvements. Master’s Thesis, Münster University of Applied Sciences, November 2008.

ABF02 M. Ajtai, R. B. Compactly Encoding Unstructured Inputs with differential compression. *Journal of the ACM*, Volume 49, Issue 3, May 2002, pages 318-367.

Mac99 MacDonald, J. Versioned File Archiving, Compression and Distribution. Technical Report, 1999.

Mad89 Madej, T. An application of group testing to the file comparison problem. *Proc. 9th Distributed Computing Systems*, June 1989, pages 237-243.

Mah05 Mahoney, M. Adaptive Weighing of Context Models for Lossless Data Compression. Technical Report CS-2005-16, Florida Tech, 2005.

MaM93 Manber, U., & Myers, E. Suffix Arrays: A New Method for On-Line String Searches. *Proc. Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, pages 319-327.

Mar79 Martin, G. Range encoding: an algorithm for removing redundancy from a digitised message. *Proc. Video & Data Recording Conference*, July 1979, pages 24-27.

Met91 Metzner, J. Efficient replicated remote file comparison. *IEEE Transactions on Computers*, Volume 40, Issue 5, May 1991, pages 651-660.

Met83 Metzner, J. J. A Parity Structure for Large Remotely Located Replicated Data Files. *IEEE Transactions on Computers*, Volume C-32, Issue 8, August 1983, pages 727-730.

MiM85 Miller, W., & Myers, E.W. A File Comparison Program. *Software: Practice and Experience*, Volume 15, Issue 11, pages 1025-1040.

MTZ03 Minsky, Y., Trachtenberg, A., Zippel, R. Set Reconciliation with Nearly Optimal Communication Complexity. IEEE Transactions on Information Theory, Volume 49, Issue 9, September 2003, pages 2213-2218.

MGC07 Motta, G., Gustafson, J., & Chen, S. Differential Compression of Executable Code. *Proc. Data Compression Conference*, March 2007, pages 103-112.

MCM01 Muthitacharoen, A., Chen, B., Mazières, D. A low-bandwidth network file system. *Proc. Eighteenth ACM symposium on Operating systems principles (SOSP’01)*, 2001, pages 174-187.

Mye86 Myers, E. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, Volume 1, 1986, pages 251-266.

Nis08 National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication 180-3, 2008.

NeW70 Needleman, S. B., Wunsch, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, Volume 48, Issue 3, 1970, pages 443-453.

Obs87 Obst, W. Delta technique and string-to-string correction. *Lecture Notes in Computer Science*, 1987, pages 64-68.

Orl93 Orlitsky. Interactive communication of balanced distributions and of correlated files. *SIAM Journal on Discrete Mathematics*, Volume 6, Issue 4, November 1993, pages 548-564.

Orl90 Orlitsky, A. Worst-case interactive communication I: Two messages are almost optimal. *IEEE Transactions on Information Theory*, Volume 36, Issue 5, September 1990, pages 1111-1126.

Orl91 Orlitsky, A. Interactive communication: balanced distributions, correlated files, and average-case complexity. *Proc. 32nd Annual Symposium on Foundations of Computer Science*, October 1991, pages 228-238.

OrV02 Orlitsky, A., & Viswanathan, K. One-Way Communication and Error-Correcting Codes. *Proc. IEEE International Symposium on Information Theory*, 2002, page 394.

Igo12 Pavlov, I. LZMA2 algorithm. URL: http://www.7-zip.org/sdk.html

Per03 Percival, C. Naive differences of executable code. Unpublished paper, URL: http://www.daemonology.net/bsdiff/, 2003.

Per06 Percival, C. Matching with Mismatches and Assorted Applications. PhD Thesis, Univerity of Oxford, 2006.

PCT11 Polk, T., Chen, L., Turner, S., Hoffman, P. Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms. RFC 6194, March 2011.

Pyn95 Pyne, C. (1995). Patent No. 5,446,888 (also 5,721,907). United States.

Rab81 Rabin, M. O. Fingerprinting by Random Polynomials. Technical Report TR-CSE-03-01, Harvard University, 1981.

RaC01 Ramsey, N., & Csirmaz, E. An Algebraic Approach to File Synchronization. *Proc. 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, pages 175-185.

RaB03 Rasch, D., & Burns, R. (2003). In-Place Rsync: File Synchronization for Mobile and Wireless Devices. Proc. USENIX Annual Technical Conference (ATEC'03), 2003.

Chr91 Reichenberger, C. Delta storage for arbitrary non-text files. *Proc. 3rd international workshop on Software configuration management*, June 1991, pages 144–152.

Ris76 Rissanen, J. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, Volume 20, Issue 3, May 1976, pages 198-203.

Riv90 Rivest, R. (1990). The MD4 message digest algorithm. RFC 1186, IETF.

Riv92 Rivest, R. (1992). The MD5 message digest algorithm, RFC 1321, IETF.

SWA03 Schleimer, S., Wilkerson, D.S., Aiken, A. Winnowing: Local algorithms for document fingerprinting. *Proc. ACM SIGMOD international conference on Management of data*, 2003, pages 76-85.

San72 Sankoff, D. Matching Sequences under Deletion/Insertion Constraints. *Proc. National Academy of Sciences*, Volume 69, Issue 1, January 1972, pages 4-6.

SBB90 Scharwz, T., Bowdidge, R. W., Burkhard, W. A. Low Cost Comparison of File Copies. Proc. 10th International Conference on Distributed Computing Systems, May 1990, pages 196-202.

ShK10 Shapira, D., & Kats, M. Bidirectional Delta Files. *Proc. Data Compression*

Conference DCC-2010, 2010, pages 249-258.

ShS03 Shapira, D., & Storer, J. A. In-Place Differential File Compression. *Proc. Data Compression Conference DCC-2003*, 2003, pages 263-272.

ShS04 Shapira, D., & Storer, J. A. In-Place Differential File Compression of Non-Aligned Files With Applications to File Distribution, Backups, and String Similarity. *Proc. Data Compression Conference DCC-2004*, 2004, pages 82-91.

SXW10 Sheng, Y., Xu, D., & Wang, D. A Two-Phase Differential Synchronization Algorithm for Remote Files. *Proc. 10th International Conference on Algorithms and Architectures for Parallel Processing*, 2010, pages 65-78.

Ski06 Skibinski, P. Reversible data transforms that improve effectiveness of universal lossless data compression, PhD Thesis, Department of Mathematics and Computer Science, University of Wroclaw, 2006.

STA03 Starobinski, D., Trachtenberg, A., Agarwal, S. Efficient PDA Synchronization. IEEE Transactions on Mobile Computing, Volume 2, Issue 1, January 2003, pages 40-51.

SuM02 Suel, T., & Memon, N. Algorithms for Delta Compression and Remote File Synchronization. K. Sayood, *Lossless Compression Handbook*, ISBN-13: 978-0126208610, January 2003.

SNT04 Suel, T., Noel, P., & Trendafilov, D. Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks. *Proc. 20th International Conference on Data Engineering*, 2004, pages 153-164.

TRL12 Tarkoma, S., Rothenberger, C., & Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *Surveys & Tutorials*, IEEE Volume 14, Issue 1, 2012, pages 131 - 155.

Tic84 Tichy, W. F. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, Volume 2, Issue 4, November 1984, pages 309-231.

Tri99 Tridgell, A. Efficient Algorithms for Sorting and Synchronization. PhD Thesis, Australian National University, February 1999.

Tri08 Tridgell, A. Rsync 3.0.0 what's new. URL: http://rsync.samba.org/ftp/rsync/src/rsync-3.0.0-NEWS

TrM96 Tridgell, A., Mackerras, P. The rsync algorithm. Technical Report TR-CS96-05, Australian National University, 1996.

Ukk85 Ukkonen, E. Algorithms for Approximate String Matching. *Information and Control*, Volume 64, Issue 1-3, January 1985, pages 100–118.

Ukk95 Ukkonen, E. Constructing Suffix Trees On-Line in Linear Time. Proc IFIP 12th World Computer Congess on Algorithms, Software, Architecture – Information Processing, 1992, pages 484-492.

IMS05 Utku Irmak, S. M. Improved Single-Round Protocols for Remote File Synchronization. *Proc. INFOCOM'05*, 2005, pages 1665-1676.

VMG07 Välimäki, N., Mäkinen, V., Gerlach, W., Dixit, K. Engineering a Compressed Suffix Tree Implementation. Journal of Experimental Algorithms, Volume 14, December 2009, article no. 2.

ChT04 Chauhan, V., Trachtenberg, A. Reconciliation Puzzles. *Proc. Global Telecommunications Conference GLOBECOM'04*, November 2003, pages 600-604 (volume 2).

WaF74 Wagner, R. A., & Fischer, M. J. The String-to-String Correction Problem. *Journal* *of the ACM*, Volume 21, Issue 1, 1974, pages 168-173.

Wei73 Weiner, P. Linear Pattern Matching Algorithms. *Proc. Switching and Automata Theory*, 1973, pages 1-11.

YIS08 Yan, H., Irmak, U., Suel, T. Algorithms for Low-Latency Remote File Synchronization. *Proc. 27th Conference on Computer Communications INFOCOM 2008*, April 2008, pages 156-160.

Yao79 Yao, A. C. Some complexity questions related to distributive computing. *Proc. 11th Annual ACM Symposium on Theory of Computing*, 1979, pages 209–213.

You06 You, L. (2006). Efficient Archival Data Storage, PhD thesis, University of California, 2006.

ZYR12 Zhang, H., Yeo, C., & Ramchandran, K. VSYNC: Bandwidth-Efficient and Distortion-Tolerant Video File Synchronization. *IEEE Transactions on Circuits and Systems for Video Technology*, Volume 22, Issue 1, January 2012, pages 67-76.

ZhX94 Zhang, Z., Xia, X.-G. Three messages are not optimal in worst case. *IEEE Transactions on Information Theory*, Volume 40, Issue 1, January 1994, pages 3-10.

ZiL77 Ziv, J., Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Volume 23, Issue 3, 1977, pages 337–343.