Random Access Archives for Efficient Compression of Many Small Files

Robert Jan Hensing

Bachelor Thesis at Utrecht University
July 8, 2011

Abstract

Collections of random-accessible small files do not compress well with existing software because the redundancy between the files is not used. When random access is required, the state of an adaptive compression algorithm must be known at the start of the file. The default state is not suitable.

A generic solution is presented to this problem, which works for a range of compression algorithms: a generated file called 'soup' is prefixed to each file to initialize the model and is later stripped from the compressed data. An algorithm that generates a soup is described, as are the considerations for soup generation. The algorithm approximates the longest common substring problem and uses these most frequent substrings to create the 'soup' file. This file is prepended to input files in order to initialize the compression algorithm. The soup is then trimmed from the compressed data.

This archiving method performs similar to solid compression storage-wise in some cases and also has the advantage of random access. An e-book was compressed to half its *compressed* size by introducing a soup.

A free software implementation will be available.

Contents

List of Figures

1 Introduction

1.1 Adaptive compression

Adaptive compression algorithms perform well on various inputs because of their flexibility. These algorithms create a model of the data while compressing and decompressing, which is used to determine the next output symbol. The model adapts to the file, making the compression algorithm adapt to the kind of data it processes, even when the characteristics of the file change at some points in the file.

An example of such an algorithm is LZ77 [5], which replaces redundant parts in its input by pointers to earlier occurrences. It adapts simply by reading its input and hence being able to reference relevant data.

Adaptive modeling has, however, been claimed not to be suitable when random access into the data is required, for example in full-text retrieval systems [6]. This is also due to the fact that the model is built or modified while reading, causing the first part of the file not to be modeled accurately. In some applications, the first part comprises the whole file.

This document challenges that claim, and proves that existing adaptive compression algorithms can be used for such applications, thereby extending the benefits of adaptive compression into the domains where random access is a requirement. These include text retrieval systems and back-up, archiving and distribution of source code, web pages, etc.

1.2 Assumptions

This document will not focus on the specifics of any application, but will use the terminology from filesystems and archiving. The matter presented here can be applied easily to database systems or other applications.

Also, it is assumed that for random access, quick access to the start of a short file suffices. This assumption does not weaken the problem: if access at random positions in a large file is demanded, such a file can be split into smaller chunks that are compressed like small files. Hence "file chunk" would be more accurate, but for clarity the word "file" is used.

1.3 Overview

In Section 2, the current methods of archiving and compression are discussed. Section 3, presents the new archiving method and Section 5 provides an empirical comparison of this and prior methods. Section 6 concludes this document. The data sources used in the measurements are described in the appendix.

2 Current Methods

2.1 Archiving

Currently, a collection of small files can not be compressed efficiently while preserving random access. Two methods of archiving can be distinguished: "solid compression" and "individual compression". Both have advantages and disadvantages.

Solid compression The files are interspersed with their metadata, like file names and sizes, and then compressed in one go. This has the advantage that the compression algorithm can take advantage of the redundancy between files. The great disadvantage is that random access is not possible. The whole archive must be decompressed.

File metadata may be stored in another location.

Example: GZIP-compressed tar (`.tar.gz`, `.tgz`); commonly used for software distribution on UNIXlike platforms.

Individual compression Each file is compressed individually and afterwards, the compressed files are stored in the archive file. Random access is possible because decompression starts at the beginning of the desired file, but the redundancy between files is left unused, as for each file the algorithm's state is reset.

Example: ZIP [4] (`.zip`, `.jar`, OpenDocument); general archiving and applications that benefit from random access.

To improve these methods, either of those may be used as a base for further improvement. In the case of solid compression, this implies adding a random access (or "seeking") capability to the compression algorithm. This can not be done with algorithms that employ adaptive modeling, because the state of the model is determined by possibly all previously read data. Even though the information contained in the model is always less than the previously read data, it is usually not viable to store the entire model frequently. Different approaches exist, but are either algorithm-specific or involve resetting the model's state.

The approach taken here borrows from the individual archiving method. The problem to solve in this case is how to get the model in a more useful state, without consuming more storage in total.

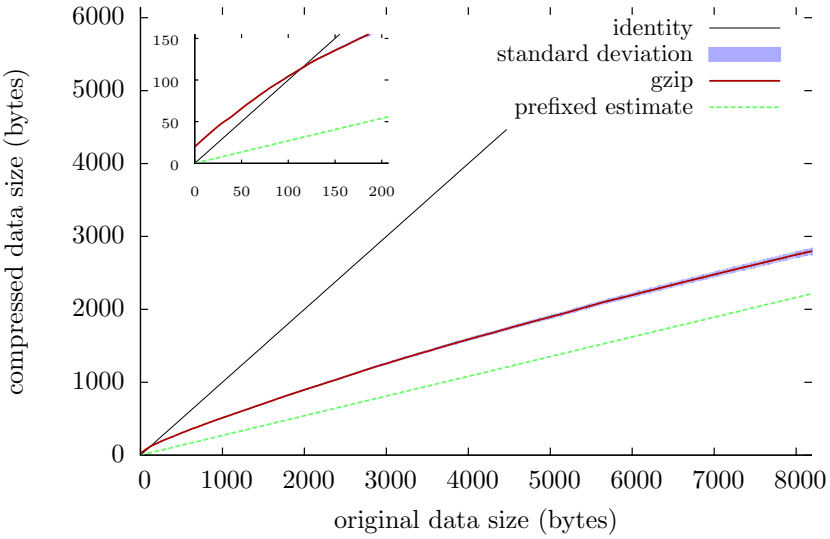
2.2 Compression of Short Files

Figure 1 demonstrates that adaptive compression algorithms like GZIP [1] need space to build their models: near the start of the file, the curve is steep, indicating a bad compression ratio[1]. Towards the end of the 8KiB file, the curve straightens, indicating that the compression ratio does not improve. This means that the model does not improve much anymore. The dashed green line shows the theoretical compression performance that could be attained if the model of a similar file were used at the start of the file. This line sets the goal for the archiving method that will be described later.

Figure 2 and 3 show that other data, an x86_64 ELF executable, may not benefit much from a better initial model. The model becomes slightly better in the first 1000 to 1500 bytes, but does not improve afterwards. Probably, the redundancy in the instruction set is modeled well, but large patterns remain undetected because the semantics can not discovered by the algorithm. Note that LZMA [3] – a more advanced algorithm – performs slightly better, but does not model large patterns either.

[1] Compression ratio is defined as the division of compressed size by the original size. Lower is better.

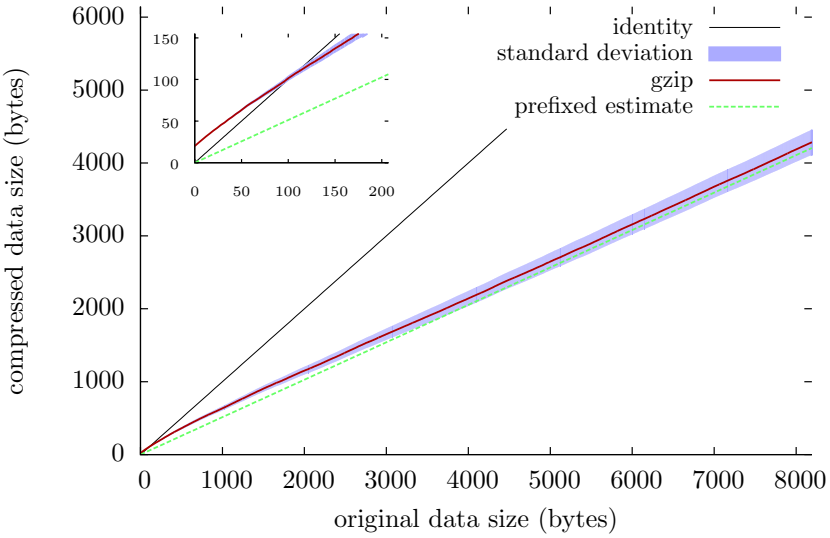
Figure 1: Compression Effectiveness on Small Inputs



Data source: "linux-kernel-c-8K", see Appendix A for details.

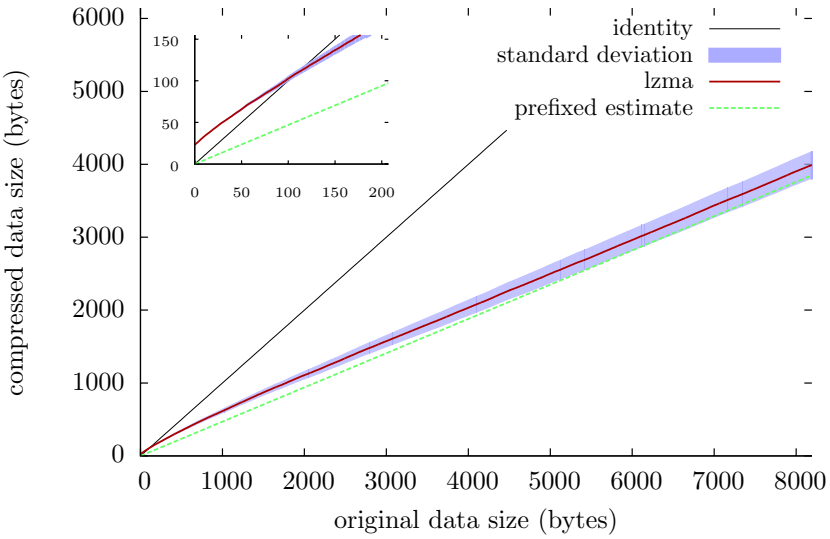
Theses figures were generated by compressing the first part of every file in a collection of files, up to the byte number on the horizontal axis. The compressed output was measured and aggregated by computing the average and standard deviation.

Figure 2: Short Model Building on Object Code (GZIP)



Data source: "gnu-tar-x86_64-8K", see Appendix A for details.

Figure 3: Short Model Building on Object Code (LZMA)



Data source: "gnu-tar-x86_64-8K", see Appendix A for details.

3 Proposed Archiving Scheme

To bring the model in a more useful state, the the redundant patterns from the input files are extracted into a file called the 'soup'. This soup is prepended to the input before compressing, so the soup builds a model that is more useful than the 'empty' model of the compression algorithm. This will result in more efficient compression of the input file. The compressed soup is then removed from the compressed files. This process is illustrated in Figure 4. However, the soup will have to be stored redundantly if the following requirement is not met:

> There exists a small constant k, such that for any soup s and file f, $c(s)$ equals $c(sf)$ up to $\#c(s) - k$ bytes.
>
> Definitions:
>
> $c(x)$**:** compression of byte sequence x
>
> $\#y$**:** the number of bytes in y
>
> sf**:** concatenation of s and f

Essentially, this means that during compression, the output does not depend on too many unread bytes. This property makes it possible to re-use the compressed soup for multiple compressed files.

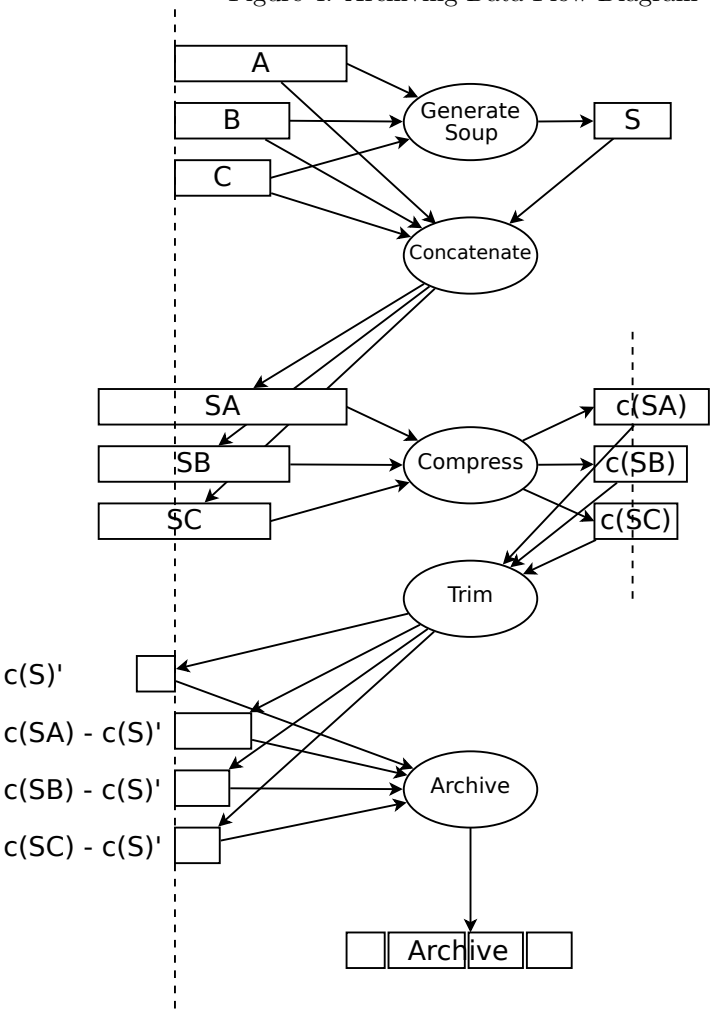
Note that this is not a formal requirement for correctness. If the requirement can not be proved for an algorithm, but holds in general, it can still be used, because when the exceptional case occurs, "individual compression" could be used.

If this requirement is not met, the state of the model will have to be stored and restored, which is algorithm-specific. Some compression algorithms do not have a suitable model. For example, algorithms that employ the Burrows-Wheeler Transform [2] are hard or impossible to adapt, because large blocks need to be permutated for the transform to do its work well.

The scope of this document will be limited to algorithms that do satisfy above property.

Figure 4 illustrates the archiving process. It works like individual compression, but also a soup is generated, which is prepended before compression and removed before storing. When the archive is read, the compressed soup is prepended to the compressed file and removed after decompression.

Figure 4: Archiving Data Flow Diagram



A, B and C are input files. The dashed lines represent the start of these files. Juxtaposition means concatenation. c(x) means compression of x. The archiving process is reversible.

4 Soup Generation Algorithm

4.1 Design requirements

The soup generation algorithm was designed to target compression with LZ77-like algorithms: algorithms that replace redundant patterns by references to earlier occurrences. In this context, the following can be percieved:

Counting: When comparing utility of including particular strings in the soup, only the number of files in which these strings occur should be considered. The total number of occurrences is not relevant, as the second and further occurrences in the same file will be references to the first.

Locality: Also, as most implementations favor references at a short distance, the most frequent substrings should be stored near the end of the soup.

Frequency: The soup should contain all substrings that occur in at least two files.

Unicity: Any substring should be contained only once. Otherwise, the dictionary will be clobbered with duplicate substrings.

Utility: Any short substring in the soup should occur in a least two input files.

As they have been worded, some of these guidelines contradict. Also, for some inputs and algorithms, they are inaccurate or not optimal. They do however clarify the goals of the algorithm.

Due to the contradictory nature of these guidelines, optimality can not achieved – at least not for a somewhat compression-agnostic algorithm.

4.2 Approximating Longest Common Substrings

The Frequency guideline hints at including the longest common substrings (LCSs) that occur in at least two files – that is: the LCS "l_1", the LCS "l_2" after removing every occurrence of l_1 (for Unicity), the LCS "l_3" after removing instances of l_1 and l_2, $\cdots$. A simple example:

> Considering the input strings `AaaaBbbbCccc` and `BbbAaCccccc`, the longest common substrings are:
>
> 1. `Cccc`
> 2. `Bbb`
> 3. `Aa`

These LCSs should be ordered, in accordance with Locality, so the frequency of each LCS must be recorded.

For the implementation, a limited depth trie[2] data structure was chosen to store the contents of the input files. Only limited depth was considered necessary, because approximate LCSs will be preserved. This will be shown later.

[2] A trie is a tree with nodes carrying single characters/bytes as values, and in which a string is represented by a path from root to leaf.

The alternative that was considered, but discarded is the suffix tree. This kind of tree is similar to the trie, but differs because a node can contain more than one character. An alternative would be a generalized suffix tree data structure, which is similar to a trie, but can store more than one character in a node. This is a more efficient representation if only few or highly redundant files are loaded, which is not the case. The real disadvantage is the added complexity of multiple characters per node. Generalized suffix trees were not entirely ruled out. They may provide a slight advantage, but this was not considered important enough.

The substrings from all files can be accounted for in a single trie, by increasing an integer variable "count" whenever duplicates are merged. This has two benefits. The first and obvious benefit is memory use and scalability. The second is more subtle: the nodes that are right before the leaves carry valuable information. If substrings of length k are stored in the trie, then for every substring of length $k-1$, the next character can be predicted by going to the corresponding node at depth $k-1$. The count values of the children of this node contain the probabilities of the corresponding characters. By looking up the next character multiple times, longer strings can be constructed, that are likely te be longest common substrings. They may not be, but for this purpose, it is just as sufficient.

4.3 Data Structure and Algorithm

Listing 1: Trie data structure

```
class Trie {
  Map<byte, Trie> children
  int merge_count = 0
}
```

Map<**byte**, Trie> is a data structure that can be used to look up Trie structures that correspond to a byte. It is injective: there is only one Trie for each key. Instead of the `count` variable suggested earlier, a variable named merge_count is used, which serves the same purpose: count = merge_count + 1.

The injective property of Map will be used to make sure each substring is counted once per file (as demanded by Counting); when constructing a trie from a file, the merge_count is always set to 0. Due to the Counting requirement, an extra trie is used when loading a file.

When two Tries are available, either by reading files or merging other tries, they can be merged. The result of a merge is a Trie whose leavas are the union of the input Tries' leaves with appropriate merge_count set.

Listing 2: Trie Merge

```
Trie
merge(Trie a, const Trie b) {
  // add children from b to a, merging when necessary
  for (key in b.children.keys()) {
    if (a.children.hasKey(key))
      // both Tries contain the key. merge will increase
      // the merge_count
      a.children[key].merge(b.children[key])
    else
      // the key is only in b, so copy it to a
      // and don't touch the merge count
      a.children[key] = b.children[key]
  }

  // add and increase the merge count
  this.merge_count = this.merge_count + 1 +
                     other.merge_count;
  }
  return a
}
```

When all files' Tries are merged, the merge_count represents the number of times the string is duplicated, equal to the number of occurrences minus one.

Note that, as the merge operation is commutative (if rewritten in a non-destructive fashion), it can be executed in parallel to a large degree.

At this point, there is one Trie containing all substrings and their frequencies. To turn them into soup, another algorithm is required to stitch these substrings together.

The algorithm can be split up in several aspects.

prediction The algorithm starts at the most frequent substring of greatest length (k) and tries to append the most probable character until the most probable next character has a merge count equal to 0. It is then stored in a set called result.

cutoff This process is cut off whenever a cycle (repetition of length k) is detected, because otherwise this part of the algorithm may produce an infinitely long output of repeated substrings. In this case also, the string is stored in result.

reverse prediction After this is done for the most frequent substring, it is done for the second-most frequent substring. However, when it is detected that the tail of length k also occurs somewhere in result, the prediction is also cut off and an attempt is made to prepend the unique part of the current string to the appropriate string in result.

This is effectively a reverse prediction. This is necessary because prediction starts at the most frequent substring, which may be part of a longer, less frequent substring.

Leaving out the reverse prediction, the algorithm is roughly as follows:

Listing 3: Simplified Soup Generation Algorithm

```
final int maxDepth
byte[] generateSoup(byte[][] files) {
  Trie merged = new Trie
  // makeTrie constructs a Trie of substrings with a
  // length of maxDepth
  // Construct the Trie, here: sequentially
  for (file in files)
    merged.merge(makeTrie(file))

  List<byte[]> longStrings =
              merged.stringsOfLength(maxDepth)
  longStrings.sortByFrequency()

  List<byte[]> output = new Set
  Trie inOutput = new Trie

  while (mystring = longStrings.pop()) {

    // don't bother at all if it's already in there
    if ( not inOutput.has(mystring)) {
      byte[] last = mystring.tailOfLength(maxDepth)

      // extend mystring until cutoff
      while (not inOutput(last)) {
        key = last.tailOfLength(maxDepth - 1)
        try{ mystring += merged.predict(key) }
        catch{ break }
        inOutput.insert(last)
        last = mystring.tailOfLength(maxDepth)
      }
      output.prepend(mystring) // not append: Locality
    }
  }
  byte[][] o = output.orderBy(compareLength);
  return concatenate(o);
}
```

The Trie.predict method simply returns the key that has the highest merge count.

last is a sliding window at the end of mystring, that is used to check for the cutoff conditions. key is 1 shorter than last, and is used for prediction. output is the list of strings that will be included in the soup. inOutput is used to quickly determine if strings are already in output.

The actual algorithm also checks why last is already in the output and attempts to merge mystring with one of the output strings (reverse prediction). This was omitted for clarity.

The time complexity of the algorithm is $O(kn + n \log n)$ where n is the number of input bytes and k is the maxDepth. The space complexity is $O(kn)$ in worst case. This complexity applies to files of random or apparently random data. This kind of input can be detected and skipped.

5 Comparison

An implementation of the described archiving method, "HAR", a dull initialism of *H*ensing *Ar*chive, will be available.

Figure 5 shows a comparison of the three possible archiving methods on the "rfcmeta" data source. This data was gathered with an early prototype. File names and locations were excluded from the measurements, because each archiving method must deal with it somehow. This run of HAR was done with seven soups: one for each range of RFC numbers $1000n \cdots 1000(n+1)-1$. Note that BZIP2is not included in the *har* compression method, because it does not conform with the requirement at page 7.

A fully functional prototype has been used on an EPUB e-Book of Dutch tax law. The results were impressive:

EPUB format: 448130 bytes (based on ZIP, which is also deflate, but slightly different from GZIP)

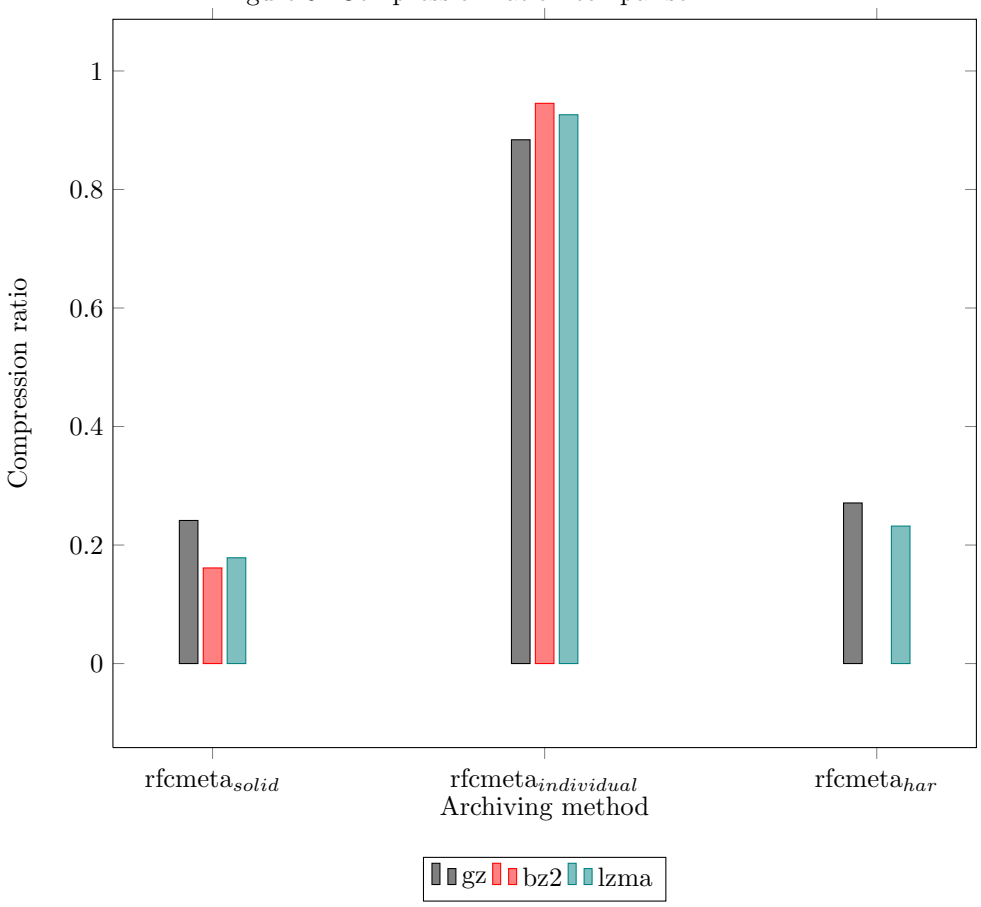
Har format, no soup: 392168 bytes (GZIPand more efficient metadata storage)

Har format, depth limit 8: 236754 bytes

Har format, depth limit 16: 214138 bytes

However, the soup did not help compressing the source package of the prototype. The cause of this deficiency remains unknown.

Figure 5: Compression ration comparison



6 Conclusion

6.1 Results

The problem of inefficient compression of short files has been identified and understood. A solution is presented for cases where many such files are found together, which makes it possible to save some context in a file called a 'soup': a generated file is prepended to the input files, so the algorithm can make 'assumptions' about the short input, based on this file. This approach works for a range of algorithms.

The design considerations of a soup generation algorithm have been elaborated and an implementation has been shown. Storing the 'soup' pays off: a prototype shows that compression ratios that are only slightly higher than those of solid compression can be achieved on very short files. An e-book was compressed to half the original format and slightly more than half of the har format without using a soup.

6.2 Opportunities and Unsolved Issues

Archives containing different types of files may benefit from multiple soups. File extensions could be used as a heuristic, but clustering will probably match more accurately.

Only the surface has been scratched as far as the soup generation is concerned. Specific algorithms may call for specific soup generation methods and the problem may be solved with very different techniques. Also, the current algorithm and its implementation can be analyzed more thoroughly and can be improved.

Some compression algorithms' implementations may be modified to inject an initial state more efficiently.

The benefit of using a trie, described at page 10 is probably not used to the fullest. Such a trie, or maybe even a part of it, is a very useful start for Prediction by Partial Matching compression algorithms. So, the trie can surely be used in another way, but will it pay off?

References

[1] P. Deutsch. Rfc1952: Gzip file format specification version 4.3. *Internet RFCs*, 1996.

[2] M. Nelson. Data compression with the burrows-wheeler transform. *Dr. Dobbs Journal*, 9, 1996.

[3] I. Pavlov. Lzma software development kit. http://www.7-zip.org/sdk.html.

[4] PKWARE. .zip file format specification. http://www.pkware.com/documents/casestudies/APPNOTE.TXT, 1989.

[5] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.

[6] Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *Software - Practice & Experience*, 25(8), 1995.

A Data sources

gnu-tar-x86_64-8K

These files were created by splitting the GNU tar executable from Ubuntu 11.04 into 8 KiB blocks, dropping the last block, which is smaller than 8 KiB.

linux-kernel-c-8K

These files were taken from the `kernel` directory of Linux 2.6.38. Files smaller than 8 KiB were removed from the experiment beforehand, resulting in 72 files of at least 8 KiB in size.

rfcmeta

This data source was created by splitting up the rfc index found at `ftp://ftp.rfc-editor.org/in-notes/rfc-index.txt`. The header was removed and the files were split up, using the RFC number as file name. The RFC number was stripped from the file. The total size is 1 107 913 bytes, divided into 6150 files. The average size is 180 bytes and the standard deviation is 24 bytes.