Design and Evaluation of a Compressed File System
Project Final Report
15-712 Software Systems

Jun Gao, Sanjay Rao, Peter Venable
(jungao, sanjay, pvenable@cs.cmu.edu)

1 Introduction

In this project, we consider issues involved in the design of a compressed file system. We propose algorithms for structuring a compressed file, and analyze the tradeoffs each make with regard to space and time efficiency. We hypothesize that no single algorithm would prove satisfactory for all possible file access patterns. We implement each of these algorithms at the user level, and evaluate their performance with workloads representing different sets of access patterns to validate our hypothesis. We also integrate all these algorithms into a single hybrid system that allows the user to select an appropriate algorithm for the access patterns he expects on each individual file, thus optimizing performance.

2 Algorithms for structuring compressed files

We present a few algorithms for structuring compressed files and discuss the tradeoffs each makes regarding the amount of metadata that needs to be maintained, the amount of space the compressed file occupies, and the time to access the compressed file. The algorithms are presented from the perspective of user-level implementation. They can be modified for better performance in a kernel implementation. For further discussion, see Appendix A.

In the following discussion, *Logical File* refers to the user's view of the file, and *Physical File* refers to the file in the underlying filesystem (eg. ext2fs) in which the compressed representation of the data in the logical file is stored. *Metadata* refers to the metadata that needs to be maintained over and above what a normal Unix filesystem maintains. A *Change* to a logical file refers to an operation that changes its contents without changing its size (e.g. replacing a character).

The algorithms are as follows:

1. The Physical file is obtained by directly compressing the entire logical file. The advantage is that this approach leads to maximum space efficiency, as there is maximum compression, and there is no overhead of maintaining metadata. The disadvantage is that an update, or a random access to a part of a large file requires dealing with the entire file.

2. The Logical file is split into fixed-size *Logical Blocks*. Each of these is compressed, and the compressed chunks are organized *contiguously* in the Physical File. The metadata required is a per-logical-block entry that keeps track of where the compressed chunk corresponding to that logical block is present in the physical file. This approach is efficient for random access of a large file. However, a change made to a logical block of a file may change the size of the corresponding compressed chunk, and this requires explicit copying of the rest of the file.

3. The logical file is split into logical blocks, each of which is compressed into chunks that are initially maintained contiguous in the physical file. Further, each chunk terminates with a "rest

of the logical block" pointer (which is empty initially). When any change to the logical block causes an increase in the size of the corresponding compressed chunk, space corresponding to the increase is allocated at the end of the file, and the "rest of logical-block" pointer is updated to point to this new allocated space. Note that repeated increases in size could lead to a logical block being represented by a chain of discontiguous chunks. When a change to a logical block causes a decrease in the size of the corresponding compressed chunk, the extra space is not freed but could be reused to accommodate future increases of the same chunk. During non-busy hours, the physical file may even be *cleaned offline*, to get rid of fragmentation and to ensure that each logical block is arranged contiguously rather than as a chain of non-contiguous blocks. This approach requires per-logical block metadata overhead that keeps track of where in the physical file the particular logical block was at the start of processing, as well as the current size of the logical block after compression.

4. The Logical file is split into fixed-size Logical Blocks. The Physical file is viewed as consisting of a set of fixed-size *physical blocks*. Each logical block is compressed; a certain amount of *forced space* is added to it and it is mapped onto *consecutive* blocks of the physical file. Note that the extra space reserved for a logical block is more than the minimum forced space added due to physical block alignment. The amount of forced space added is the same for all logical blocks of a given file, but it may vary from file to file depending on an indication given by the user. This algorithm requires a per logical-block metadata overhead that keeps track of where in the physical file the particular logical block is and the number of physical blocks that the logical block occupies. It is not highly space efficient due to internal fragmentation. A change in size to a compressed chunk due to a change made to the corresponding logical block of the file could in most cases be accommodated by the extra space available in the last physical block of that chunk, thus avoiding explicit recopy of the file. However, explicit recopy cannot be avoided when the change of size requires a change in the number of physical blocks to be allocated.

We summarize the trade-offs between individual algorithms in Figure 1.

2.1 Metadata

Each file begins with a generic header identifying the algorithm used and the logical and physical size of the file. Each of algorithms 2,3 and 4 requires a per-logical-block entry. For algorithms 2 and 4, all metadata is at the start of the file. There are a small number of direct entries for any file, and on demand larger blocks of metadata entries are allocated that can accommodate entries for say k logical blocks. An explicit recopy in these algorithms involves updating of all metadata entries, and having the entire metadata at the start is more efficient. The disadvantage of this is that during an append of a large file, a new metadata block needs to be created for every k logical blocks created, which in turn requires recopying of the file. In algorithm 3, we dealt with per-logical-block entries in a similar fashion to how UNIX inodes deal with disk-block addresses. That is, there are slots for a few direct entries, and a few indirect entries. The direct pointers point to real data blocks and the indirect pointers point to a metadata block which contains pointers pointing to real data blocks. These indirect pointer blocks are kept mixed with data blocks, instead of being grouped together at the beginning of the file, as in algorithms two and four.

3 Software Design

3.1 Description

The system in which we compare various algorithms for structuring compressed files is a "transparent" filesystem for linux. By "transparent" we mean that once the filesystem is mounted, it can be treated like any other mounted filesystem. Ordinary UNIX commands, such as ls, cp, cat, or rm, plus most existing software, work as expected, which has the bonus of making the system a prototype

No.	BRIEF DESCRIPTION	METADATA STORAGE	SPACE EFFICIENCY	TIME EFFICIENCY
1.	Compress or Decompress entire file.	No extra storage.	Highly efficient.	Good for small files, and sequential access of large files. Bad for random access, append or change to large files.
2.	Compress or Decompress logical blocks of file, store contiguously.	Per Logical block entry that tracks where the logical block maps to in the compressed physical file.	Less efficient than (1) as compression of smaller chunks.	Good for small files, sequential access , random access and update of large files. Bad for change to large files - recopy.
3.	Compress or Decompress logical blocks of file, store contiguously. "Rest of logical block" pointer is maintained for each compressed chunk. If change in logical block leads to increase in size of compressed chunk, create space at the end corresponding to the size-increase and update the "rest of lb pointer". Any space freed due to a decrease in size is reused to accomodate future increases. Also, clean offline.	Per logical block entry that tracks where the logical block mapped to in the compressed physical file at the start of a run., and the current size of compressed logical block. also, pointer space for each fragment of the compressed chunk corresponding to each logical block.	Slightly worse than (2) offline. Potentially suffers from internal fragmentation online, due to shrinkage, but this is likely to be small.	Similar to (2) for all kinds of accesses, except for change where much more efficient.. Also, may suffer from slight performance loss for reads due to parts of a logical block being discontiguous.
4.	Map each compressed logical block onto integral number of consecutive physical blocks, forcing some space for each logical block. Causes internal fragmentation.	Per logical block entry that tracks where the logical block maps to in the compressed physical file, as well as the number of physical blocks occupied by that logical block.	Worse than (2) due to internal fragmentation. Worse than (3) too.	For changes, usually similar to (3). However may be as bad as (2) if explicit recopy needed.

Figure 1: Algorithms for Structuring Compressed Files.

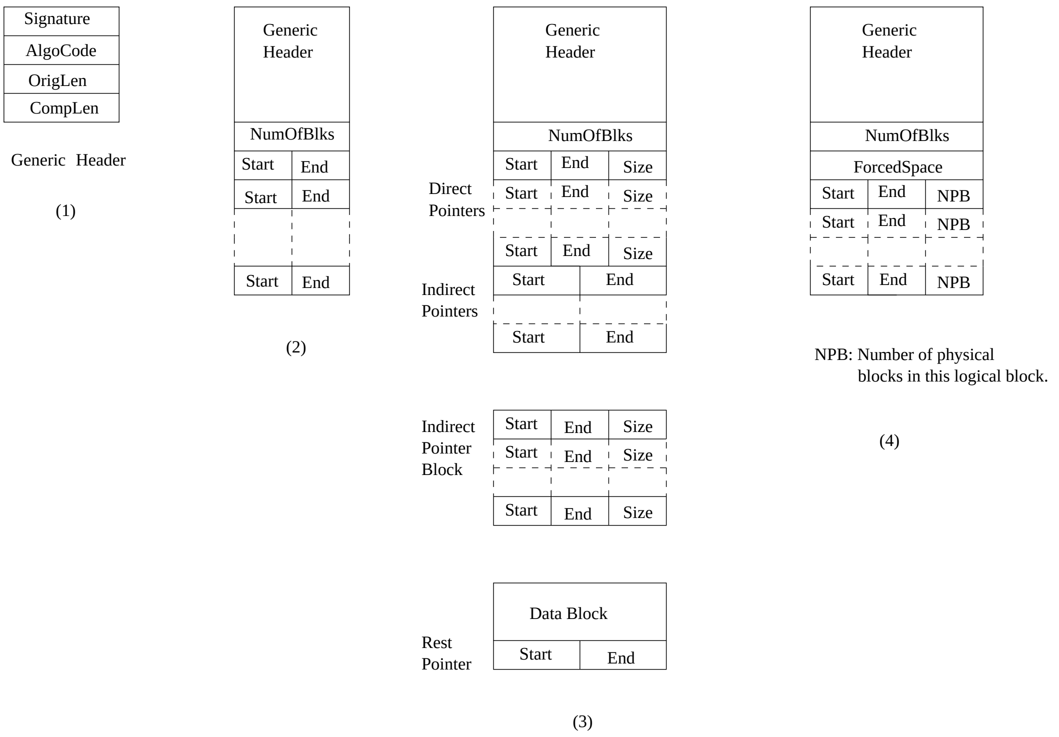

Figure 2: Metadata used in algorithms 1-4.

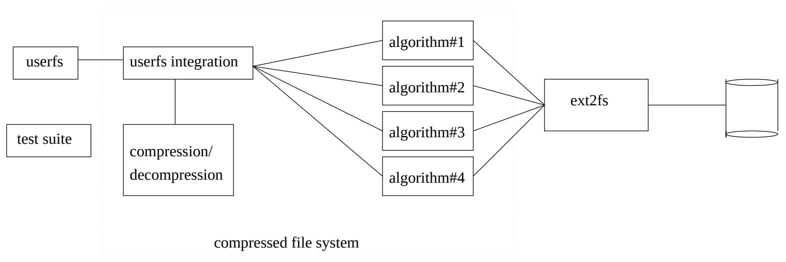

Figure 3: Module Decomposition

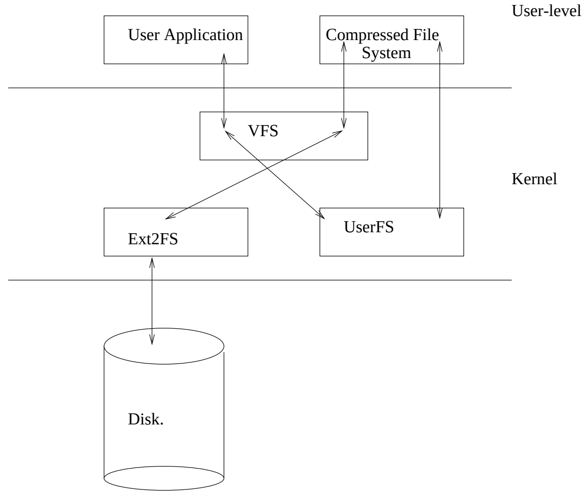

Figure 4: Filesystem Access Data Path

for a truly general-purpose compressed filesystem. Of course, in this experimental system, some less crucial features of transparency are not implemented, but they could easily be added in a production system. Similarly, this prototype is implemented with the crutch of *userfs*; in a production system speed would be greatly enhanced.

3.2 Module Decomposition

Module Descriptions

1. *userfs integration* - Handles filesystems calls which are passed though *userfs*. Provides a clean and small interface to the compression/decompression module and to each algorithm.
2. *compression/decompression* - Compresses and decompresses data. Handles blocks either individually or in a stream.
3. *algorithms* - Each algorithm described above will be implemented separately, conforming to the same interface to the userfs integration module.
4. *test suite* - The test suite will apply the workloads described below and gather cost information for analysis and evaluation.

Figure 3 illustrates the relationships between these modules.

3.3 UserFS

Our implementation is on a Linux platform. We use *userfs 0.9.2*. Userfs is a mechanism that allows a normal linux user process to be a filesystem. This needs to be loaded into the kernel as a module.

Once a userfs module is mounted to a directory, such as /mnt/compressed, any filesystem accesses referring to files within that directory tree are sent to the userfs module. In order to implement a compressed filesystem, we wrote a module to handle all the major filesystem calls, compressing and decompressing along the way. When a system call requests data through userfs, our module looks up the physical file corresponding to the virtual file in the mounted directory, processes metadata to find the correct part of the file, decompresses the data, and passes it through userfs to the application which made the (ordinary) system call. The system works similarly for other operations, such as writing to a file. Figure 4 traces the path of a system call.

3.4 Hybrid System

As an application of this knowledge, we have implemented the compressed filesystem with all four algorithms available in a hybrid fashion. That is, files may be individually specified as to which algorithm they should use, so the best algorithm for each file can be chosen, depending on its specific workload.

The user specifies which algorithm to use by simply typing a prefix to the filename when creating the file. For example, to use algorithm 2 when creating a file named myfile, use a filename like /mnt/compressed/algorithm2:myfile. The forced space for algorithm 4 may be specified in a similar fashion. For example, to use algorithm 4 with 128 bytes of forced space per logical block, use a filename like /mnt/compressed/algorithm4:128:myfile.

This system also makes it very simple to do the "offline cleaning" recommended for files which become fragmented after many updates. To clean a file, simply copy it to a temporary file, delete the original, and recopy the the temporary file back to the old name. This causes the file to be freshly written with no fragmentation.

4 Evaluation Methodology

4.1 Hypothesis

Particular algorithms for structuring compressed files should work better than others for specific file access patterns, and different ones should work better for different workloads. We verify this with empirical results and demonstrate which algorithms are appropriate for which workloads, and investigate how much difference the choice of algorithm makes.

4.2 Test Circumstances

The test software is a set of simulations written by ourselves, and therefore easy to control. It repeatedly runs the same set of tests (including various workloads) using various algorithms. The results of the test consist of time and space measurements. The space measurements are easy to observe by simply inspecting the files.

For time measurements, we use the system real-time clock, purportedly accurate to one microsecond, which is well in excess of the precision needed for our measurements, which are usually in the tens of seconds. We chose real time to insure that all relevant time expenses are included, such as system time, userfs time, I/O time, etc. The downside of this is that other system processes can affect the results. We minimize this by running the final tests on a minimal system, that is, with no other work going on.

Since the test software involves some use of randomization, the random number seeds are regulated so that the exact same test is used for each algorithm. That way the results of each test run are directly comparable across algorithms. In different test runs, different sets of random numbers are used.

4.3 Workloads

The following list of file access patterns has been selected from a wide range of possible patterns as representative workload set. Each workload is annotated with a real-world example it represents. We had considered the possibility of testing a much wider range of workloads, but concluded that it is better to have a small set of realistic workloads representing familiar tasks than a more complete set that includes many loads rarely encountered in practice, which would result in a proliferation of data and obscure the results. Hence, these representative workloads should suffice:

1. Sequential read of a large file (e.g. grep).
2. Many appends of a very large file. For example, consider applications that monitor and log traffic passing a network. The log could grow very large, recommending compression, but frequent updates must be handled efficiently.
3. Random rewrites of records, interleaved with sequential reads (Eg. Database applications).

4.3.1 Files used

A mention of the files that we used is in order, as the compression process depends highly on the nature and content of the file, and could have an important bearing on the results seen. The files used are summarized in the table below. In future, we refer to these files by their file sizes. An important limitation in our evaluation is that the biggest file we used was only 4 MB, and potentially more interesting results could have been obtained with larger files.

File Size	Description
20K	Text File containing mainly alphabets, and a few digits.
130K	Text File containing mainly alphabets, and a few digits.
1MB	A file that contains a list of several IP addresses.
4MB	A file that has a log of several Network packets.

5 Results and Analysis

We present and discuss the results obtained for each of the workloads on every algorithm. Algorithm 0 refers to an algorithm that does no compression, and is used as a baseline for comparison. (It is implemented so as to experience the overhead of userfs, except that it directly accesses an uncompressed version of the file).

5.1 Sequential Test

This test was conducted by sequentially reading a file 100 times. Reported below is the total time for the 100 sequential reads, and the percentage compression obtained ($(Origsize - Compsize) * 100/Origsize$).

Sequential						
	Time(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB
0	0.35	2.10	107.21	0	0	0
1	2.95	15.19		50.26	60.38	
2	10.35	73.49	1966.51	44.59	47.66	69.63
3	10.39	73.67	1944.88	42.16	47.42	69.30
4	10.41	73.47	1959.37	41.28	43.58	66.60

As expected, Algorithms 2,3,4 perform very similarly in terms of time-efficiency while Algorithms 2 and 3 do slightly better than Algorithm 4 with regard to space-efficiency. Also as expected is

Algorithm 1's better space efficiency as compared to the other algorithms. While we expected Algorithm 1's time efficiency to be slightly better than that of other algorithms, we were surprised to find the remarkable improvement of a factor of five. This may be explained due to our optimized implementation of Algorithm 1. We wanted to avoid decompressing the entire file for the read of each block during a sequential read of the file. To do this, we maintained enough state with a file being read, so that if it was identified that the file is being sequentially read, then we could carry on the decompression process from where it had stopped in the previous read. However, the previous read call may have caused more data to have been decompressed than needed by that call itself; To ensure that any meaningful optimization is derived, we decided to cache this extra data. Consequently, we were using a cache of $8KB$. Note that this caching is inherently essential for the good performance of the algorithm, and even a modest cache of $8K$ could mean a tremendous performance improvement for the algorithm. No doubt, a good performance improvement could be expected using caching in the other algorithms as well.

5.2 Append Test

In this test, we started off with a file size 0, and appended data constantly till it reached the sizes shown in the Table below. Each append operation itself appended 10 random words from a collection of about 300 English words, as well as some redundant information that was common across all appends. After each append the file was flushed on to disk. The Table below reports the file-sizes created, the compression ratio achieved and the total time for the final file to be created after all the appends.

Append						
	Time(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB
0	0.08	0.49	16.86	0	0	0
1	53.41			61.45		
2	10.68	74.85	2573.77	54.06	53.69	55.59
3	10.68	75.78	2613.59	42.96	44.62	46.51
4	10.50	74.60	2486.90	49.90	47.74	49.89

Algorithms 2 and 4 seem to perform better than Algorithm 3 with regard to space efficiency. The reason for this is that repeated appends to Algorithm 3 causes each logical block to be represented as a chain of consecutive fragment, each fragment having the associated overhead of the next fragment pointer. However, we believe that this shortcoming can be overcome in a future implementation by modifying Algorithm 3 so that if there is an increase in size of the compressed chunk corresponding to that logical block which owns the last fragment in the physical file, then, the size of the last fragment is increased, rather than a new fragment created. Algorithm 2 is slightly better than 4 because it has no reserved space at the end of each physical block.

Algorithms 2,3,4 perform similarly in terms of time efficiency. Algorithm 1 is horribly slow in this instance, because each change to the file (each append) requires the decompression, and subsequent recompression of the entire file, resulting in $O(n^2)$ time cost (where n is the number of appends). This is so bad that we didn't even bother measuring algorithm 1 on large files.

5.3 Database Test

Each of the test files was considered to consist of records of size 64 bytes. A change operation was implemented by reading some random record (64 bytes) of the file, and modifying it. The modified record consisted of 64 randomly generated letters of the English Alphabet. Usually the modified record was more random than the initial record, and would increase the size of the compressed file. The Database test itself consisted of 10 runs, each run involving 50 changes

to the file, and 1 sequential read. The file was opened before and closed after each of the 10 runs. The table below presents the total time for the 500(10*50) changes, and total time for the

10 sequential reads. The compression percentage values give the compression percentages after the changes have been made.

Database Test									
	Change(sec)			Read(sec)			Compression(%)		
Algorithm	20K	130K	4MB	20K	130K	4MB	20K	130K	4MB
0	0.16	0.25	0.59	0.03	0.21	7.49	0	0	0
1	270.93			0.33			30.96		
2	53.81	61.94	45.94	1.30	8.20	177.77	23.97	37.31	68.94
3	54.35	61.69	31.89	1.36	8.31	178.59	11.28	34.29	68.52
4	53.66	61.46	34.12	1.31	8.23	198.00	19.54	33.92	65.85

Algorithm 1 has the best compression; Algorithm 2 has better compression than Algorithms 3 and 4. Between Algorithm 3 and 4 however, the space efficiency is better for Algorithm 3 for larger files, while it is better for Algorithm 4 for smaller files. There is wastage of space in Algorithm 4 due to internal fragmentation; On the other hand heavy changes to a small number of logical blocks (as with the 20K file which had 3 logical blocks and 500 changes), would cause each of these logical blocks to consist of a long chain of fragments in Algorithm 3, each fragment having some space wasted for the rest of the chain pointers. For larger files however, it is more unlikely that the same logical block is so heavily changed, and secondly the pointer overhead becomes more insignificant.

When we consider change-times, Algorithm 1 is extremely slow, and we deemed it sufficient to show the poor performance for a small file. Algorithm 3 is almost 33 % faster than Algorithm 2 for changes to the 4 MB file. This is because it avoids explicit recopy of the file, that is needed for every change in Algorithm 2. Algorithm 3 is slightly faster than Algorithm 4, because of the occasional recopies required for Algorithm 4. For smaller files, the change times of the three algorithms is almost indistinguishable; this may due to the low costs of explicit recopy of a small file.

While the main purpose of this test was to analyze the change-times, we also measured the sequential-read times mainly to see if there was any loss of performance in sequential reads with Algorithm 3. We found that the loss of performance is marginal, if at all; more surprisingly, there seems to be a slight deterioration of performance for Algorithm 4. We believe that the good performance for Algorithm 3 is probably explained by the fact that the fragmentation is very light for the larger 4 MB file; for the smaller files good values are seen in spite of heavy fragmentation possibly because of a read-ahead mechanism in the underlying filesystem (ext2fs). We do not have a satisfactory explanation for the poorer performance of Algorithm 4: one possible explanation is that the compressed file is larger in size, and the extra time is due to the reading of a bigger file.

5.3.1 Algorithm 4 Parametrization

All our experiments involving Algorithm 4 so far involved a forced-space of 0. In this section, we study the effect of varying the forced space. Of the operations read,append and change, the only operation where we expect a difference in performance is change and we focus only on this. In this case, the test files were viewed as consisting of records of size 256 bytes. A change operation involved reading a random 256 bytes, and replacing it with a set of random alphabets. The test itself involved 10 runs, each run consisting of 100 change operations. The file was opened before and closed after each run. Presented below is the total time taken for all the (10*100) changes, the compression rates before any change was made, and the compression rates after all the changes were made.

Algorithm 4						
	Time(sec)		Compression(%) Initial		Compression(%)Final	
FS	1MB	4MB	1MB	4MB	1MB	4MB
0	76.33	80.70	74.23	66.59	51.98	60.69
64	76.85	79.56	74.08	65.71	51.92	60.49
256	75.61	73.28	74.08	63.20	51.92	59.90

Our initial expectations were that increasing the forced space would lead to poorer space performance, but better time performance. While our observations seem to indicate the trend, it is not as pronounced as we had expected. For the other files we tried ($< 1MB$), the variation in forced-space caused almost no difference in performance.

6 Conclusions and Future Work

In this project, we considered issues involved in the development of a compressed file-system. We specifically focussed on the problem of structuring a compressed file in this system. We have built a hybrid system, that supports different algorithms for structuring compressed files, and have shown that particular algorithms perform better with particular access patterns. The user has the choice of indicating which algorithm he would like to use based on the access patterns likely to be seen on the file. Based on our evaluations, we make the following recommendations:

1. Sequential Read - Algorithm 1 with minimal caching works very well; Further it gives the best compression ratio. The cache required is small (8K), and is critically essential for good performance of the Algorithm.

2. Appends - Algorithm 2 works well being as time-efficient as Algorithms 3 and 4, yet having a much better space efficiency.

3. Changes - Algorithm 3 seems to be the best candidate. It evidently outperforms Algorithms 1 and 2. Our evaluation shows its performance to be slightly better than Algorithm 4 in terms of time efficiency; It is better than Algorithm 4 with regard to space efficiency (except for small Files); Finally, it has the benefit that it can be cleaned offline, which is not available to Algorithm 4. We had feared a slight loss in performance for sequential access; however such a loss was not revealed in our evaluations. Further, our evaluations revealed only a marginal improvement on forcing free space in Algorithm 4. On the balance, it is fair to choose Algorithm 3 over Algorithm 4 in most cases.

While our system functions as a prototypical compressed filesystem, demonstrating a basic amount of transparency and showing the relative merits of various algorithms, it is not really practical for regular use. It is too slow to compete with other filesystems, and it doesn't support all operations, such as symbolic links. While it would be relatively simple to extend its functionality to include all the regular filesystem operations, the efficiency can only be improved so much while the system resides in user space, passing all calls through the extra userfs layer. A production version of this system should be integrated with VFS inside the kernel instead of using userfs. Not only would this avoid the overhead of multiple redirections of system calls, but it would allow some of the algorithms to be improved through direct access to the underlying disk layout mechanism and would enable metadata to be kept seperate from the data along with the traditional UNIX inode.

A look at the various tables shows us that the time-efficiency of all our algorithms is disappointing when compared to the efficiency of an uncompressed system. Note that this overhead is not due to userfs, as the uncompressed system was also allowed to experience the overhead associated with userfs. However, adopting good caching-mechanisms could help alleviate the overhead associated with compression somewhat; The remarkable performance of Algorithm 1 on sequential accesses with just an 8K cache seems very encouraging in this respect. Caching mechanisms that must be investigated include (i) Caching of Metadata on the opening of a file, as some of our algorithms require frequent updating of metadata entries; (ii) Caching of logical blocks when they are uncompressed, so that a subsequent access to the same logical block does not require the compression overhead.

Finally, in this project, we have not considered issues related to fault tolerance (e.g. If a system crashes while we are in the process of updating metadata, recopying file etc. then the file contents are badly destroyed). However, this question is orthogonal to the design of a compressed file system itself, and methods normally employed in regular file systems could be used.

7 Relationships to Course Material

The main focus of this project was on comparing and improving compressed file layout algorithms, which took place in the context of the development of a compressed filesystem. Clearly, this is directly related to the class topic of filesystems. In addition, at a high level our investigation is directed at identifying situations where each of several competing techniques is effective and trying to support all of them, rather than discover one globally best technique. This approach has been motivated by several class readings (e.g. Munin).

A Kernel Implementation

We discuss how the algorithms described earlier may be implemented at the kernel level one by one:

1. Can be implemented as is.

2. Metadata is added to the usual contents of each UNIX inode. Per logical block entries may be organized in the fashion of direct, indirect and doubly indirect blocks. An optimization is that the per-logical-block entry need only contain the byte in the compressed file where the corresponding compressed chunk starts.

3. Metadata is added to the usual contents of each UNIX inode, adding doubly indirect pointers.

4. Metadata is added to the usual contents of each UNIX inode. Per logical block entries may be organized in the fashion of direct, indirect and doubly indirect blocks. The block size of the physical file may be fixed as the physical disk block size. An explicit recopy of data can be avoided, by having only a recopy of inode pointers.

B Compression Algorithm

We have chosen the Lempel-Ziv-Welch (LZW) algorithm[1] to compress and decompress data. LZW was developed by Terry Welch in 1984 for hardware implementation in high performance disk controllers by refining an earlier algorithm published by Lempel and Ziv in 1978. This algorithm belongs to the category of dictionary methods in data compression, which utilizes the property of many data types containing repeating code sequences. Text files and image files are two good examples of such data type.

The LZW method is very popular in practice (e.g. image GIF format, UNIX compress utility) and its major advantage over other dictionary methods (e.g. LZ77) is its speed, because the number of string comparisons is significantly less.

The LZW algorithm creates a dictionary of the phrases that occur in the input data. When an encountered phrase is already present in the dictionary, only the index number of this phrase will be sent to the output.

This method also does not need to read in the whole input data to perform the compression which gives no limitation on the file size, i.e. the file length can be much larger than the available memory size.

The algorithm itself contains two parts: Encoding and Decoding.
Notation: P = current prefix, C = current character.

B.1 The Encoding Algorithm

```
1. At the start, the dictionary contains all possible roots, and P is empty;
2. C := next character in the charstream;
3. Is the string P+C present in the dictionary?
      a.if it is, P := P+C (extend P with C);
      b.if not,
              i.output the code word which denotes P to the codestream;
             ii.add the string P+C to the dictionary;
             iii.P := C (P now contains only the character C);
4. Are there more characters in the charstream?
         if yes, go back to step 2;
         if not:
              i.output the code word which denotes P to the codestream;
             ii.END
```

[1]LZW algorithm is a patent of Unisys. (US Patent No. 4558302). Free use of the method is allowed except for the producing of commercial software

B.2 The Decoding Algorithm

```
1. At the start the dictionary contains all possible roots;
2. cW := the first code word in the codestream (it denotes a root);
3. output the string.cW to the charstream;
4. pW := cW;
5. cW := next code word in the codestream;
6. Is the string.cW present in the dictionary?
         if it is,
              i.output the string.cW to the charstream;
             ii.P := string.pW;
             iii.C := the first character of the string.cW;
             iv.add the string P+C to the dictionary;
         if not,
              i.P := string.pW;
             ii.C := the first character of the string.pW;
             iii.output the string P+C to the charstream and add it to the
                dictionary (now it corresponds to the cW);
7. Are there more code words in the codestream?
         if yes, go back to step 4;
         if not, END.
```

References

[1] K. Sayood, *Introduction to Data Compression*, pp.97-116, Morgan Kaufmann Publishers, Inc., 1996.

[2] Data Compression Reference Center, *Compression Algorithms*, http://www.rasip.fer.hr/research/compress/index.html, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.

[3] Jeremy Fitzhardinge, *Userfs 0.9.4.2*, http://sunsite.anu.edu.au/archives/linux/ALPHA/userfs/userfs.lsm