An Automatic System for File Compression

T. RAITA
Department of Computer Science, University of Turku, SF-20500 Turku, Finland

This paper presents a compression system for disc files. The system is automatic in the sense that once started, it selects those files from the user directory which have not been recently used, compresses them, builds a directory for them and places the data as a whole under one filename. This has resulted in savings of approximately 50 % in a DEC2060 computer system and in university use, which is due to the coding and to the space saved in avoiding the allocation of unnecessary directory pages of the file system. Alternative ways to design a compression system are also discussed.

Received May 1985

1. INTRODUCTION

Text compression, although for many years researched and discussed,[2, 4, 13, 14] has been neglected almost everywhere as a potential tool in a data manipulation environment. Only simple techniques, such as null or blank suppression, have been implemented as minor enlargements to operating systems or database management systems.[1, 6, 13].

There are reasons for this reluctance; for example, the usage of compressed data might get complicated if no (standard) compression technique has been in advance agreed upon, or if the end user has to wait too long for compression or/and decompression. Also, the rapid growth of data on a disc seems to continue no matter what actions we do to slow it down.

There are, however, certain situations where data compression is worth considering:

(*a*) when the data are static in nature, that is, the most frequent operation is the insertion of new data and

(*b*) when the available storage space is small. This concerns specially microcomputers and the like, where the capacity of a diskette determines what kind of application programs one can run.

The characteristics in (*a*) are often found in an information retrieval system; for example, in a library application,[5, 12, 13] where a huge amount of reports, abstracts, titles, etc., is stored. This, as well as the limited capacity of a microcomputer,[7, 8, 15] has resulted in the consideration of a coding system.

It is also worth considering compression when the performance of a time-sharing system (with hundreds of users and thousands of files) begins to deteriorate as a consequence of a small free disc space. An alternative for compression (excluding buying more equipment) is migration, where the files which have been unused for a long period of time are archived from disc to tape. This is acceptable at least for some files but there are always programs and data files for which the migration only causes unnecessary data transmission between two different storage levels.

Also, an individual end user is usually not willing to archive his files voluntarily because the retrieving of a file is not under his own control and may take a long time, depending on the operator's work load. Therefore the user regards the archiving of a file almost the same as deleting a file. Thus, storing rarely but periodically used files in compressed form on the disc seems to be a flexible way both from the computer system's and the user's point of view.

A general-purpose compression system is described in Section 2. The system scans automatically the files on the disc (or drum), selects the unused ones and compresses them. The system is adaptive in the sense that it does not use a fixed code table but forms for each file a table of its own on the basis of the structure of the data.

In Section 3 some results obtained with the system in our computing environment are presented. Different ways to build the system are discussed in Section 4, and in Section 5 some further ideas are presented.

2. THE DESCRIPTION OF THE SYSTEM

In order to keep a minimal amount of data on the disc and to avoid superfluous complexity in the migration operation, a compression system was designed and implemented. The aim was to make a flexible system for an end user to code and decode his files. The two most important requirements for the system were stated as

(*a*) the decoding of a file should be fast and

(*b*) the compression gain achieved should be high.

The coding time was not considered to be of great importance, because coding does not demand any activities from the end user and is therefore suitable to be carried out as a background job.

The system consists of the *coding system* and the *decoding system* (Fig 1). The coding system contains:

(*a*) a routine, which *searches* for the header blocks of the files under the given directory and determines from that data the files to be coded (program 'Search'),

(*b*) a routine, which *codes* the files selected by 'Search' into a compressed form by using an improved version of Rubin's [17] text compression algorithm (program 'Code') and

(*c*) a routine for *building* or *updating* a directory for the compressed files and *deleting* the original ones (program 'Build').

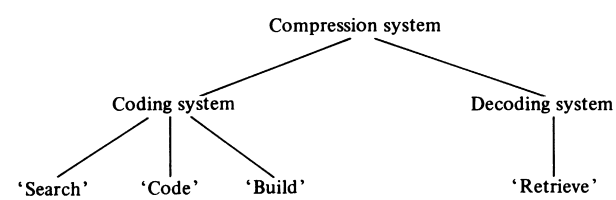

Figure 1. The parts of the compression system.

The decoding system consists of a routine for *retrieving* and *decoding* a given file and *updating* the directory and the bucket of compressed files (program 'Retrieve').

The system includes also two utility programs: the first outputs the names of the compressed files and the second compresses the files independently from the automatic system.

The system was designed so that it can be started in two ways: an end user can start it by running the appropriate routine or the operator invokes it just like the migration operation. If started by the operator, the system works as follows. The 'Search' routine scans all the files of the directory of a user, looks for the *file description blocks* (FDB) and determines the activity of each file by using the number of days since the creation of the file and the number of days since the latest non-write access (read) of the file. *If more than k days has passed since the file creation and the file has not been read during the n days before the moment the FDB is examined, the file is considered to be inactive and is included in the group of coded files.* The quantities k and n are system parameters, which are chosen to suit the particular computing environment. They can also be varied in a specific environment at different times to control the amount of disc space freed. After all inactive files have been found, they are coded one at a time and gathered under a single filename in the same directory from which the files were selected.

Figure 2 shows the layout of the compressed data. It consists of the header and the coded data. The header starts with the number of files (m) stored in the bucket. After that follow the names and the starting points of the files. Each compressed file itself consists of two different parts: the code table and the actual coded form of the data. The code table is used to expand the codes to the original character strings.

After the file bucket has been formed, the 'Build' routine deletes the original files. Then the same is done in the next directory. This is continued until all directories have been scanned.

Another way to utilize the system is to let every individual end user compress his own files any time he wants. He can choose proper parameter values every time he uses the system to suit his purposes. Moreover, the user can also explicitly name the files he wants to be compacted independently of the automatic selection process.

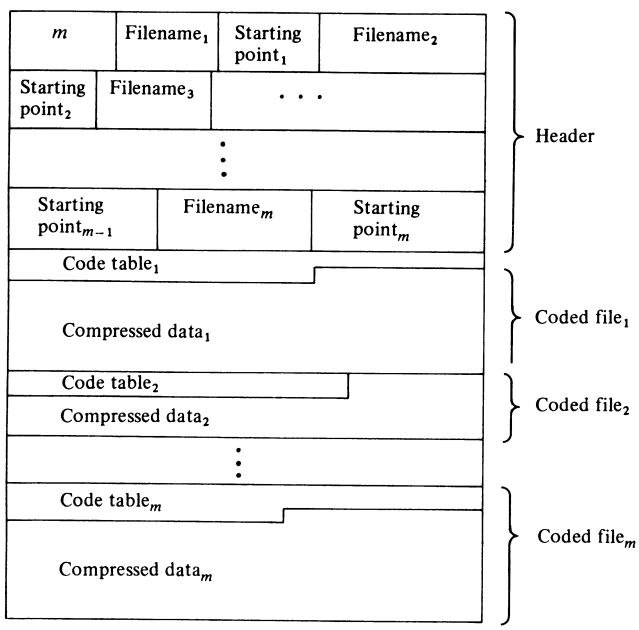

Figure 2. The structure of the compressed data.

It is obvious that the performance of the compression system depends crucially on the code routine used. The choice of the compression algorithm is determined by the type of data coded, the application area and the way the compression process is started (by the user or by the operator). Huffman coding[11] is one of the most frequently used coding schemes. In this situation, however, it was felt that its decoding time is too long. In addition, the compression gain achieved with the character-based Huffman coding is usually moderate as compared to more sophisticated methods.

A further factor which makes the use of Huffman code complicated in this context is that it codes fixed-length blocks (characters) to variable-length bit strings. If a fixed-length block is used as a basic unit of the compression system, the managing of the directory and inserting/retrieving a file from the bucket of coded files remains simple. Therefore we have chosen a modified version of F. Rubin's incremental encoding technique,[17] which codes character blocks of variable length into single characters. This leads to the restriction that we can compress only character files and not binary files – for example, program object files. Appendix 1 contains the description of the compression technique used.

The compression system needs a way to determine whether or not it is profitable to compress a certain file. In our system this decision is based on two facts: how long ago the file has been created (parameter k) and how many days have passed since the latest read access to the file (parameter n). There are no other details in the file's description block which could help us in the decision problem. Because no additional information about the files is gathered between the compression dates, we are equipped with only a minimal amount of data with which to characterise the behaviour of the file. The better values we can get for the parameters, the less coding and decoding must be done. To find good approximates for k and n, we must study the behaviour of the files.

In our computing environment, where the files of students and researchers play a central role, it is natural to assume that after a relative short period of testing and using a file, its activity rapidly decreases. This is particularly true as regards the program source files and the data files. The observations we made on a sample of 70 files (see Fig. 3) strengthen this assumption.[16] The files were chosen randomly from different user directories and they were selected to be representative of various kinds of usage. The owners of the files did not know they were observed.

From Fig. 3 we can see, for example, that during the five days from the 16th to the 20th day after the file creation, the file is read only during 0.3 days on the average. The cumulative count of read access days was calculated to be 4.0 during the first 60 days the file exists. Using the negative exponential distribution extracted from the statistics of Fig. 3 we can calculate the next reference date for the 'average' file in the sample (Fig. 4).

A single file may, however, behave very much unlike the average file. Therefore we cannot rely entirely on the

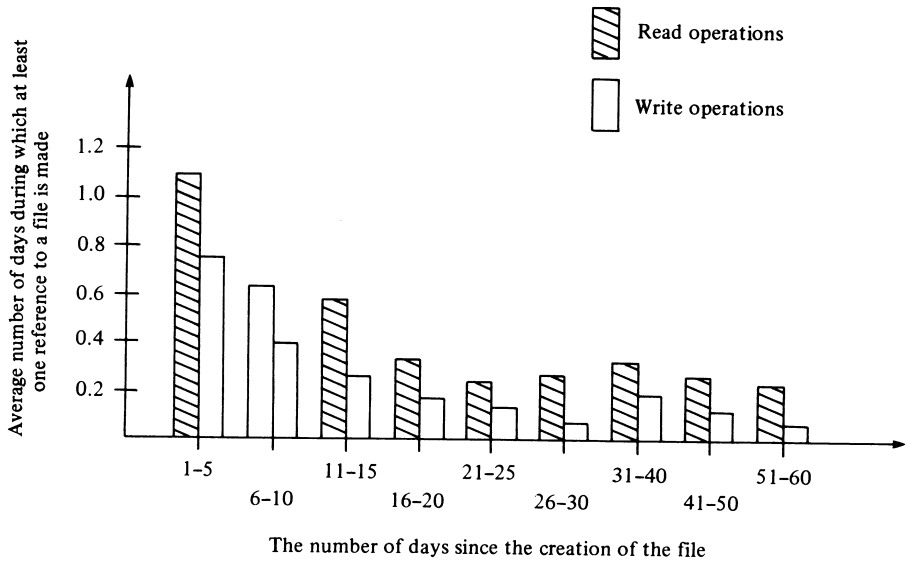

Figure 3. The number of read and write accesses as a function of the time from the creation of the file.

curve of Fig. 4 when we are determining the activity of the file. To increase the probability that the file is not needed sooner than predicted, a check is made to see whether it has been recently referenced. This means that the file is compressed *only if*

(*a*) according to the curve *g* it is profitable to compress the file *and*

(*b*) the file has not been referenced during the *n* previous days.

The conditions (*a*) and (*b*) measure the activity of a file somewhat roughly because they are based on average values taken from a sample. To make the selection more precise, the features of each file should be taken into account. This is discussed in more detail in Appendix 2.

3. THE ENVIRONMENT AND THE EXPERIMENTAL RESULTS

The DEC2060 computer system services approximately 2000 users from the six faculties of our university. Extremely heavy work load in the computer has resulted in a very weak performance, which to some extent is due to the small free disc space.

When analysing the contents of the disc area, we found out[16] that

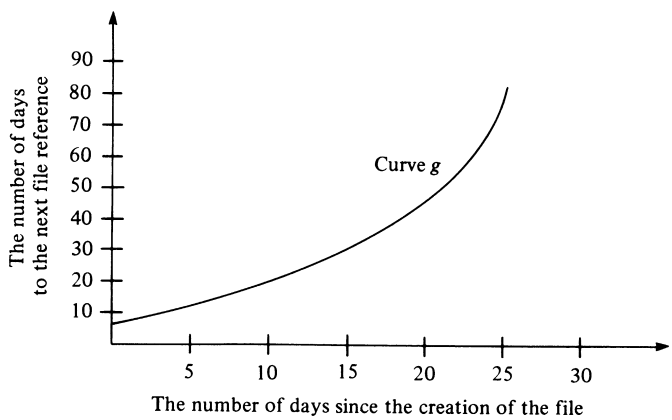

Figure 4. The dependency of the next reference to the file as a function of the time since file creation.

(*a*) almost 75% of the files were ASCII files (byte size is 7);

(*b*) small files were dominant (95.7% of all files were shorter than 20 pages, those of length one page (= 512 36 bit words) constituted alone almost 60% of all files);

(*c*) the number of read and write operations to a file as a function of time from the time of creation show a negative exponential distribution as was shown in Fig. 3.

According to (*a*) there are good reasons for using a character-oriented coding scheme. Observation (*b*) makes the efficiency of the coding algorithm seem of somewhat secondary importance, because the smaller the file, the smaller the percentual saving achieved by coding but the bigger the gain from the bucketing. On the other hand, for the large files the space is saved mainly because of the compression and the bucketing has only a minor influence on the result. Observation (*c*) indicates that there are, as expected, a lot of files which are left unused once written and used (this result is expected to vary in different environments).

Let us study how the bucketing of the files affects the space allocation. Many operating systems use a tree-structured file system in managing the user data. The DEC2060 file system is sketched in Fig. 5.

On the lowest level (leaves) are the actual data blocks. Above them reside the index blocks, the super index blocks (needed if a file is longer than 512 pages) and the directory blocks. Concentrating only on small files (with no super index block), we can see that we can always save space by putting files together, since

(*a*) unnecessary index blocks can be added to the list of free pages and

(*b*) internal fragmentation[10] which is caused by the fact that memory is allocated on a block-by-block basis, is eliminated.

This means that $m-1$ directory pages are saved in bucketing and on the average $m/2$ pages in avoiding the internal fragmentation. This yields on the average total savings of $3m/2-1$ pages.

The compression algorithm was used to code several randomly selected character files (mainly source program files) of different types and sizes. The results are given in Table 1. The modest space saving obtained with the

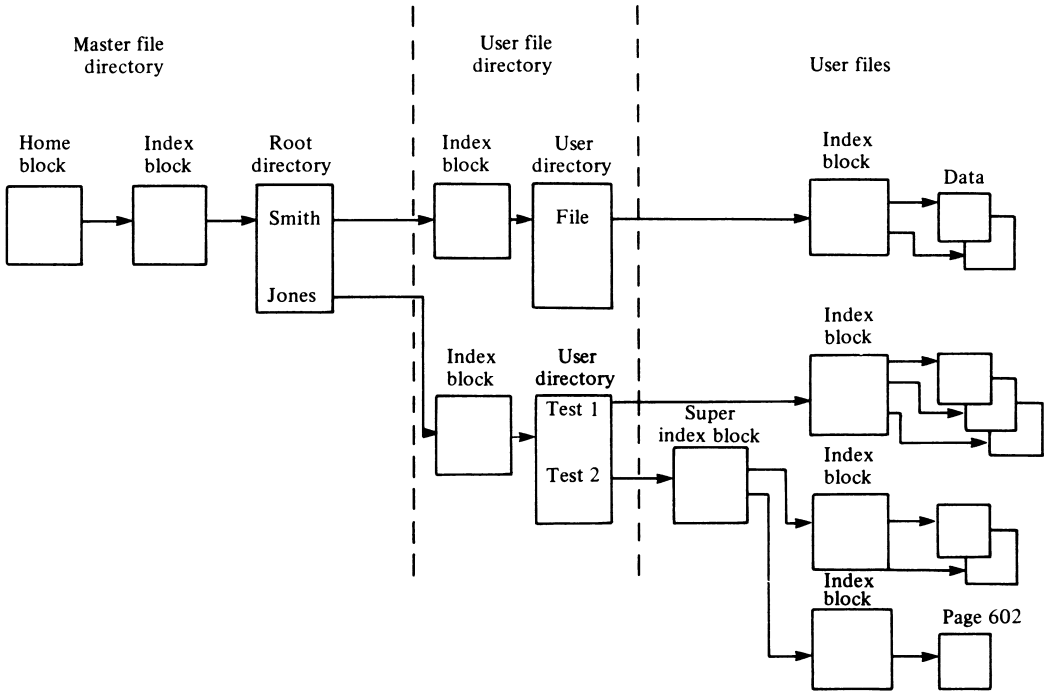

Figure 5. The structure of the DEC2060 file system[9].

Table 1. Comparison of the savings using incremental encoding and Huffman coding

Type of test file	Number of test files	The average savings (%) achieved with Incremental coding	Huffman coding
Assembler program	7	37.0	23.2
Cobol program	9	42.6	30.8
Fortran program	8	42.7	28.7
Pascal program	10	40.1	26.4
Finnish text	7	38.2	34.8
Data files	7	63.3	48.0
Average		44.0	32.0

Table 2. The total savings with bucketing and coding

S_{orig} (bytes)	m	Space needed (bytes) Before compaction	After compaction	Savings (%)
2000	5	13840	6268	54.7
	10	27680	11768	57.5
	15	41520	17268	58.4
4000	5	23840	11768	50.6
	10	47680	22768	52.2
	15	71520	33768	52.8
6000	5	33840	17268	49.0
	10	67680	33768	50.1
	15	101520	50268	50.5
8000	5	43390	22768	47.5
	10	86780	44768	48.4
	15	130170	66768	48.7

character-based Huffman coding is due to the fact that the method does not take into consideration the correlation of adjacent characters.

The observed coding/decoding times per 1000 bytes were 1.25/0.06 s for incremental coding and 0.25/0.17 s for Huffman coding respectively.

To sum up, if we had m equal size files in the bucket, then the space needed before compaction is on the average $m \times (S_{orig} + 256 + 512)$ bytes and after $m \times (1 - S_{comp}/100) \times S_{orig} + 256 + 512$ bytes, where S_{orig} is the actual size of the original file (in bytes) without the fragmentation and S_{comp} the savings (expressed in percentage) obtained with the coding technique used. Table 2 shows the total savings achieved with incremental coding.

The results of Table 2 show the relative importance of coding to the bucketing. We can see that the larger the file is, the smaller is the savings due to the bucketing. All in all, the bucketing produces only approximately 15% of the average 52% total savings and the rest is gained by the coding.

4. ALTERNATIVE DESIGN CONSIDERATIONS

Another strategy in implementing the system was to put together all the files the 'Search' routine has found and then compress the whole bucket in one pass. This has the advantage that only one coding phase must be performed and this is usually faster than coding each file separately. In addition, only one code table must be formed when the bucket is set up. The same table could possibly be utilised also when new files are inserted into the existing bucket.

The strategy has, however, some disadvantages, which cannot be easily overcome:

(*a*) the retrieving of a file becomes more complex, because the file boundaries disappear in the coding. This can be avoided, if we use special markers on the

boundaries, but it adds unnecessary complexity to the coding algorithm and the file directory,

(*b*) the code table is constructed on the basis of all files in the bucket. If the files are heterogeneous, the savings in compression may be poor. When new files are coded and appended to the bucket, the result gets still worse.

The problems in statement (*b*) can be solved in two ways. Firstly, we may decode and code the whole bucket when updating must be done. This obviously costs a lot of CPU time and therefore we should delay the deletions and insertions until time is available for it. This means that we must have some way to mark the retrieved files as logically deleted. Secondly, we can try to improve the coding result by building more than one bucket. For example, we can put all Pascal files into one bucket, COBOL files into another, and so on. This results in a new directory level and has still the disadvantages just mentioned above. Also the benefit gained from the bucketing becomes questionable, because it depends on the gain achieved by the coding. In addition, some kind of reorganisation is needed.

The compression system has been designed so that the 'nucleus' of the system, the incremental encoding algorithm, is easily changeable. If another coding technique is preferred, the module which implements Rubin's method is substituted by another. If the codes are of variable length, the directory must be organised to point to bits instead of bytes. Also, some complications might occur in this situation, if a file were to become larger when the compression routine is applied and it were therefore stored in the bucket in its original form.

In any case, the change from a character to a bit compression algorithm (see, for example, Ref. 3) would be justified, because it would make the system more general; we would be able to compact all files regardless of their byte size and more savings could be achieved.

5. OUTLINES FOR FURTHER DEVELOPMENT

The compression system could be improved/extended so that:

(*a*) Really large files could be coded by taking a sample from the file and forming the code table on the basis of the sample. An analysis should be carried out to show that the sample is representative of the original data.

(*b*) The files could be coded using bigram coding or some other quick but less-space-efficient method and standard code table(s).[7,8]

(*c*) The selection of files for compression becomes still more reliable. This could be accomplished by gathering data about references/writes to the files at regular time intervals and using the statistics to predict the next reference time,

(*d*) the coding and decoding routines are to be embedded in the existing system, rather than being used as voluntary utility programs. In microcomputers they could be placed into the operating system or into a device driver. They would always be executed before the data is transferred from the main memory (input buffer) to the output buffer (main memory) in the case the data is written (read) to? (from) the diskette.

In larger computers it seems reasonable that the responsibility for the compression and decompression should be left to an advanced disc controller.[13]

6. CONCLUSIONS

An implementation of an automatic disc file compression system is discussed. The system makes use of a coding algorithm and of the structure of the file system to get a saving of approximately 52% in disc space. The savings achieved may be regarded in some instances as only moderate while in others even 'too good', depending on the relative emphasis given to the CPU time as compared to the free disc space. The different needs of systems can be taken into consideration by adjusting the coding algorithm and the compression system parameters accordingly.

Up to this point we do not have sufficient empirical results concerning the choice of system parameters. It is obvious that there are files which the system cannot manage properly because of their exceptional usage pattern. Further research in this field is still in progress.

Acknowledgement

The author wishes to thank Dr Olli Nevalainen for many valuable comments on the manuscript.

APPENDIX I. THE DESCRIPTION OF THE CODING ALGORITHM

The incremental encoding algorithm[17] substitutes *frequently occurring substrings* by *codes*, which are characters not found from the original file. The substrings are found by iteratively calculating the frequency of character pairs and coding the pair with which most savings is achieved. This is done until all codes (or pairs for which savings are gained) are exhausted. Then the original file is coded optimally with the substrings obtained so far. If the benefit of some pairs deteriorates during the process, they are returned to their original place and the corresponding codes are freed. The whole process is repeated until no new substrings are found or all codes are in use. For a detailed presentation of the coding technique, see.[17]

The basic algorithm is very time-consuming because of the repeatedly performed character pair frequency calculation, pair substitution and optimal encoding. We have speeded up the algorithm at the expense of some loss in the compression gain. The modified algorithm selects and replaces a bucket of pairs in one scanning of the file. No pairs are returned nor is the file coded optimally. The algorithm works as follows:

(1) Find the characters not used in the file.

(2) Calculate the frequency of the character pairs.

(3) Choose the r pairs with greatest frequencies and sort them into descending order by frequency.

(4) Of these r pairs exclude those which are overlapping. That is, starting from the pair which has the highest frequency, eliminate every pair which has a common component (i.e. a character or a code) with some pair already examined.

As a result from this phase no two pairs have identical characters as their component. This avoids the situation where coding one pair precludes the coding of some other pair.

(5) From the remaining pairs, exclude those which do not bring any compression gain.

(6) Choose codes for the rest of the pairs and substitute the pairs for the codes in the file.

(7) If no more codes exist or no profitable pairs can be found, then stop. Otherwise go to 1.

The size of the bucket (that is, the number of pairs to be substituted in each iteration) can be varied using different values for r. The more we select pairs in step 3, the larger is the size of the bucket after all exclusions have been made. If the size is kept large, the fewer the buckets and the shorter the substrings found. At the same time the sorting time becomes dominant in the selection process. On the other hand, if small buckets are used, the coding time becomes long because of many passes through the file. Therefore some kind of 'optimal' size for the bucket must be found. According to the experiments we have done, an initial size of 40–50 pairs which results in the final size of 7–15 pairs per bucket has given the best performance. With these bucket sizes the coding requires 5–14 passes through the file. The last bucket is usually not full because of missing code characters. To make the algorithm work more efficiently, two solutions may be used:

(*a*) the value of r is made dependent on the number of free codes, or

(*b*) the coding is continued only if the bucket size exceeds a fixed threshold value.

The latter procedure is chosen in our implementation.

APPENDIX 2. AN ADVANCED MODEL FOR FILE SELECTION

The selection of a file for compression can be made more reliable if the characteristic features of the file are used. Let us consider a particular file and three dates associated with it (Fig. 6.).

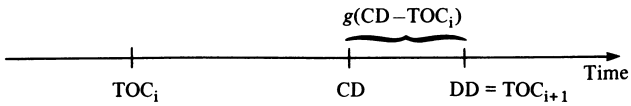

Figure 6. The basic period of a file. TOC = the time of file creation; CD = the compression date; DD = the decompression date.

TOC_i is the time of (ith, $i = 1, 2, \ldots$) file creation. After some time the file is considered to be inactive and it is coded at CD. Suppose the file is needed again at DD and it is therefore decompressed. Decompression is regarded as a recreation date of the file from the system's point of view. Therefore, $DD = TOC_{i+1}$ and the time interval from TOC_i to DD can be regarded as a cycle which is repeated until the file is removed from the disc.

Using the curve g shown in Fig. 4 we can write

$$DD = CD + g(CD - TOC_i).$$

The running time needed for coding and decoding can be calculated using the characteristics of the compression algorithm. The cost of 1 CPU second, which is largely determined by the computing environment in which we are working, is then used to express the total costs from compression. Also, the cost of storing a page of data on the disc is known. From these quantities we can (at least theoretically) determine when it is profitable to compress a file. The decision may be dependent on the behaviour of the file. The behaviour can be partly determined from the extension of the filename which is stored in the FDB. We could, for example, compress the files with standard source language extension earlier than the one with object language extension.

Let T_c and T_d denote the CPU time needed to code and decode a file and C_c the cost of using the CPU 1 s. Then the costs from the compression are $(T_c + T_d) \times C_c$.

The cost of storing the file on the disc is composed of the original and the coded form of the file (see Fig. 6). If we denote by S_{orig} and C_d the size of the file (pages) and the cost of storing data on the disc per day and page respectively and further $dt = \text{CD} - \text{TOC}_i$, the cost of storing the uncompressed file is $dt \times S_{\text{orig}} \times C_d$. During the time from CD to DD the costs are

$$g(dt) \times ((1 - S_{\text{comp}}/100) \times S_{\text{orig}} + 1/m) \times C_d,$$

where S_{comp} means savings achieved with using a certain compression technique (percentage from the original file) and m the number of files in the bucket. Therefore, when using the compression system, the total costs for the file are

$$C_{\text{comp}}(dt) = dt \times S_{\text{orig}} \times C_d + g(dt) \times ((1 - S_{\text{comp}}/100) \times S_{\text{orig}} + 1/m) \times C_d + (T_c + T_d) \times C_c. \quad (1)$$

The term $1/m$ is the file's share of the directory page needed in the file system. It is here assumed that the bucket of compressed files and the associated directory are under the same filename. Note that S_{comp} is a function of the contents of the file (see Table 1) and could therefore also have effect on the time of coding.

On the other hand, if we leave the file uncompressed, the cost during the same time is

$$C_{\text{unc}}(dt) = (dt + g(dt)) \times (S_{\text{orig}} + 1) \times C_d, \quad (2)$$

where the constant 1 is due to the directory page needed.

From equations (1) and (2) we can see that, although the characteristics of the compression algorithm are important, the structure of the file system and the page size also have a substantial influence on the total amount of space saved. In the particular case of DEC2060 and its file system, it can easily be shown that, as to the storage space, it is always worth adding a file to the bucket regardless of the space savings gained with the actual compression algorithm.

The three variables T_c, T_d and S_{comp} in equations (1) and (2) are dependent on the compression algorithm. After they have been fixed, the first day the compaction of the file is profitable is found by solving

$$C_{\text{unc}}(dt) \leftrightharpoons C_{\text{comp}}(dt). \quad (3)$$

The form of the file has an effect on the amount of savings obtained with the compression algorithm. Also, as mentioned earlier, the behaviour of the file may differ substantially from that shown in Fig. 3. Due to the fluctuations in the file's activity, the solution of equation (3) still gives us only an approximation. To get more knowledge about the usage pattern of the file, we could use a control period which is located around the earliest compression time from (3); see Fig. 7.

If the file is not read during the control time CT, it is compressed, otherwise not. The value of the CT as well as the difference between EMC and the start of the CT

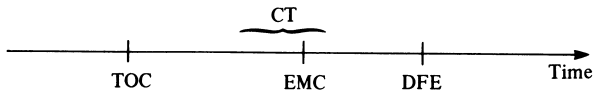

Figure 7. Important dates of a file. TOC = the time of file creation; EMC = the earliest moment for the compression to gain profit; DFE = the day when the file description block is examined (today); CT = the control time.

can be varied according to the experiences gained. Depending on the file type, the CT can be located on either side of EMC or it can be extended on both sides of it as shown in Fig. 7. Specifically, if the CT is placed in front of EMC or it is set to zero, the file is always compressed as soon as it is calculated to be profitable.

The use of the CT helps us on the next DFE to determine what caused the file to remain uncompressed. At that time we can possibly make use of another CT, which is placed earlier than DFE in time. In this way we can continually notice that the file has been active earlier and we only have to show that this assumption is still valid. The longer the file has been noticed to be active, the shorter is the time during which its activity is observed. Thus the size of the CT is a function of the difference DFE−TOC and may be made to converge to some fixed constant.

REFERENCES

1. P. Alsberg, Space and time savings through large data base compression and dynamic restructuring, *Proceedings of the IEEE*, **63**, (8), 1114–1122 (1975).
2. J. Aronson, *Data Compression – A Comparison of Methods*. Report No. NBS-SP-500-12, National Bureau of Standards, Washington, USA (1977).
3. L. R. Bahl and H. Kobayashi, Image data compression by predictive coding 11, *IBM Journal of Research and Development* **18**, 172–179 (1974).
4. A. Bookstein & G. Fouty, A mathematical model for estimating the effectiveness of bigram coding. *Information Processing and Management* **12**, 111–116, (1976).
5. A. C. Clare, E. M. Cook and M. F. Lynch, The identification of variable-length, equifrequent character strings in a natural language data base. *Computer Journal* **15** (3), 259–262 (1972).
6. H. Corbin, An introduction to data compression. *Byte*, pp. 218–250 (1981).
7. D. Cortesi, An effective text-compression algorithm. *Byte*, pp. 397–403 (1982).
8. R. H. Davis, C. Rinaldi and C. J. Trebilcock, Data compression in limited capacity microcomputer systems. *Information Processing Letters*, **13** (4, 5), 138–141 (1981).
9. DECSYSTEM-20, *Operating system, Digital Equipment Corporation, Educational Services Manual* (1980).
10. A. N. Habermann, Introduction to operating system design. *The SRA Computer Science Series* (1976).
11. D. A. Huffman, A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* **40** (9), 1098–1101 (1952).
12. M. F. Lynch, Compression of bibliographic files using an adaptation of run-length coding. *Information Storage and Retrieval* **9** (4), 207–214 (1973).
13. C. A. Lynch and E. B. Brownrigg, Application of data compression techniques to a large bibliographic database. *Proceedings of the 7th International Conference on Very Large Data Bases, Cannes, France*, pp. 435–447 (1981).
14. A. Mayne and B. James, Information compression by factorising common strings. *Computer Journal* **18** (2), 157–160 (1974).
15. M. Pechura, File archival techniques using data compression. *Communications of the ACM* **25** (9), 605–609 (1982).
16. T. Raita, A compression system for disk files [in Finnish]. Thesis for the degree of licentiate, University of Turku (1984).
17. F. Rubin, Experiments in text file compression. *Communications of the ACM* **19** (11), 617–623 (1976).