A Technique for High-Performance Data Compression

Terry A. Welch, Sperry Research Center*

Data stored on disks and tapes or transferred over communications links in commercial computer systems generally contains significant redundancy. A mechanism or procedure which recodes the data to lessen the redundancy could possibly double or triple the effective data densitites in stored or communicated data. Moreover, if compression is automatic, it can also aid in the rise of software development costs. A transparent compression mechanism could permit the use of "sloppy" data structures, in that empty space or sparse encoding of data would not greatly expand the use of storage space or transfer time; however, that requires a good compression procedure.

Several problems encountered when common compression methods are integrated into computer systems have prevented the widespread use of automatic data compression. For example (1) poor runtime execution speeds interfere in the attainment of very high data rates; (2) most compression techniques are not flexible enough to process different types of redundancy; (3) blocks of compressed data that have unpredictable lengths present storage space management problems. Each compression strategy poses a different set of these problems and, consequently, the use of each strategy is restricted to applications where its inherent weaknesses present no critical problems.

This article introduces a new compression algorithm that is based on principles not found in existing commercial methods. This algorithm avoids many of the problems associated with older methods in that it dynamically adapts to the redundancy characteristics of the data being compressed. An investigation into possible application of this algorithm yields insight into the compressibility of various types of data and serves to illustrate system problems inherent in using any compression scheme. For readers interested in simple but subtle procedures, some details of this algorithm and its implementations are also described.

The focus throughout this article will be on transparent compression in which the computer programmer is not aware of the existence of compression except in system performance. This form of compression is "noiseless," the decompressed data is an exact replica of the input data, and the compression apparatus is given no special program information, such as data type or usage statistics. Transparency is perceived to be important because putting an extra burden on the application programmer would cause

* This article was written while Welch was employed at Sperry Research Center; he is now employed with Digital Equipment Corporation.

0018-9162/84/0600-0008$01.00 © 1984 IEEE

development costs which would often exceed the value of compression. As illustrated in Figure 1, compression is viewed as being very similar to a communications channel in that it codes a sequence of bytes into an artificial format for improved physical handling and later decodes the compressed image back into a replica of the original message.

The data compression described by this model is a reversible process that is unlike other forms of "data compression"—such as data reduction or data abstraction—in which data is deleted according to some relevance criterion.

This article focuses on compression of both text and numeric data—intermixed—in commercial computer applications. Whereas previous articles have placed particular emphasis on text compression in their discussions of compression algorithms for computer data,[1-3] this article will cover more of the systems aspects of integrating a compression algorithm into a computer system with high-performance storage devices.

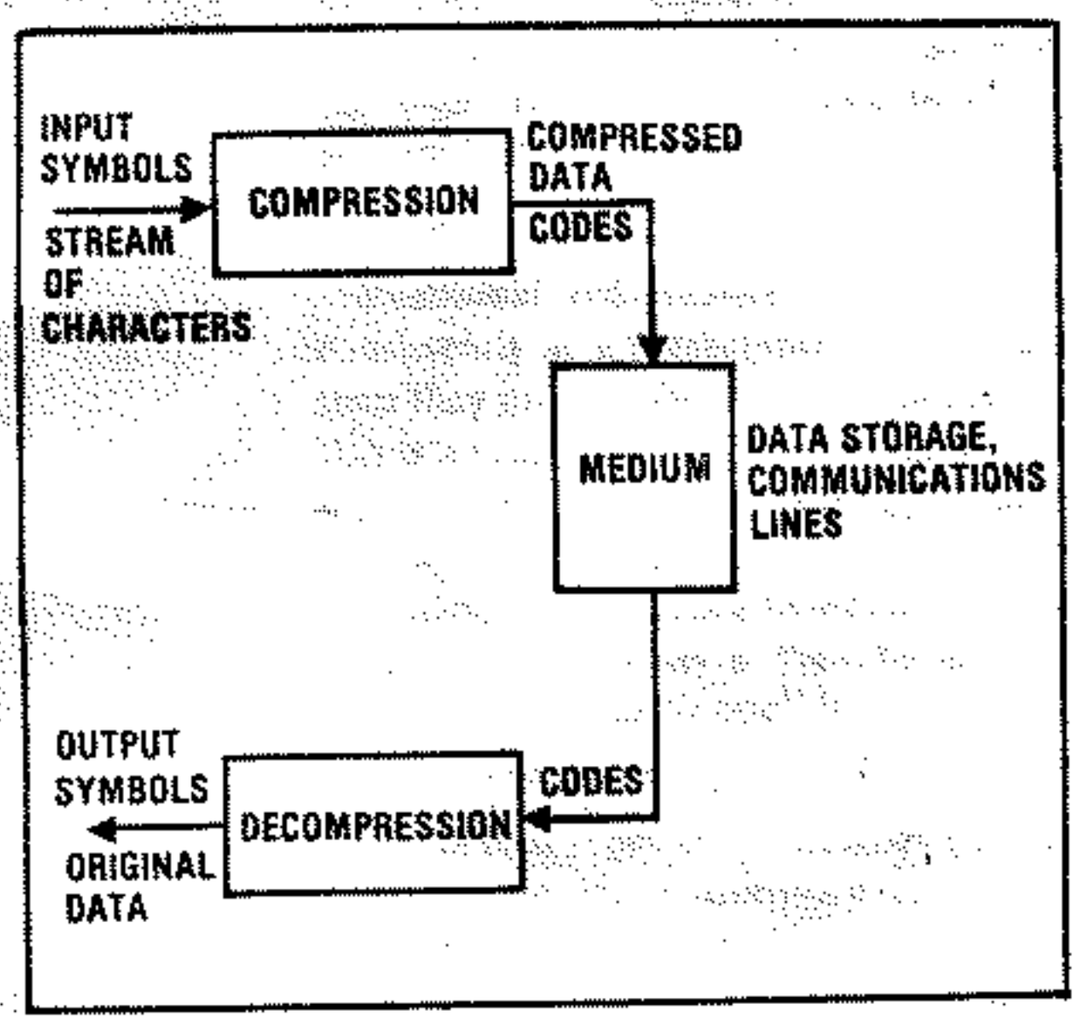


Figure 1. A model for a compression system that performs transparent compression.

Types of redundancy

Four types of redundancy can be found in commercial data. Redundancy here is confined to what is observable in a data stream (without knowledge of how the data is to be interpreted). Redundancy in the form of unused data or application-specific correlations are not considered here. For illustration, types of redundancy will be described as they might be found in two kinds of files: English text and manufacturing parts inventory records (see Figure 2). The redundancy categories described are not independent, but overlap to some extent.

Character distribution. In a typical character string, some characters are used more frequently than others. Specifically, in eight-bit ASCII representations nearly three-fourths of the possible 256-bit combinations may not be used in a specific file. Consequently, nearly two bits of each eight-bit packet might be saved, for a possible 25-percent space/time savings. In English text, the characters occur in a well-documented distribution, with *e* and "space" being the most popular. In an inventory record, numeric values are very common—the existence of binary or packed-decimal numbers can shift the statistics—and constraints on field definition can cause character distributions to vary significantly from file to file. For example, the choice of whether to use alphabetic or numeric values to identify warehouse sites can shift the distribution in the inventory file. Likewise, the extent to which descriptive text appears in the inventory records will influence the average number of bits needed per character.

Character repetition. When a string of repetitions of a single character occurs, the message can usually be encoded more compactly than by just repeating the character symbol. Such strings are infrequent in text, occurring as blank spaces in place of tabs or at the end of lines. However, in formatted business files, unused space is very common. An inventory record will frequently have strings of blanks in partially used alphabetic fields, strings of zeros in high-order positions of numeric fields, and perhaps strings of null symbols in unused fields. Graphical images, especially the line drawings of business graphics, are mostly composed of homogeneous spaces, and their

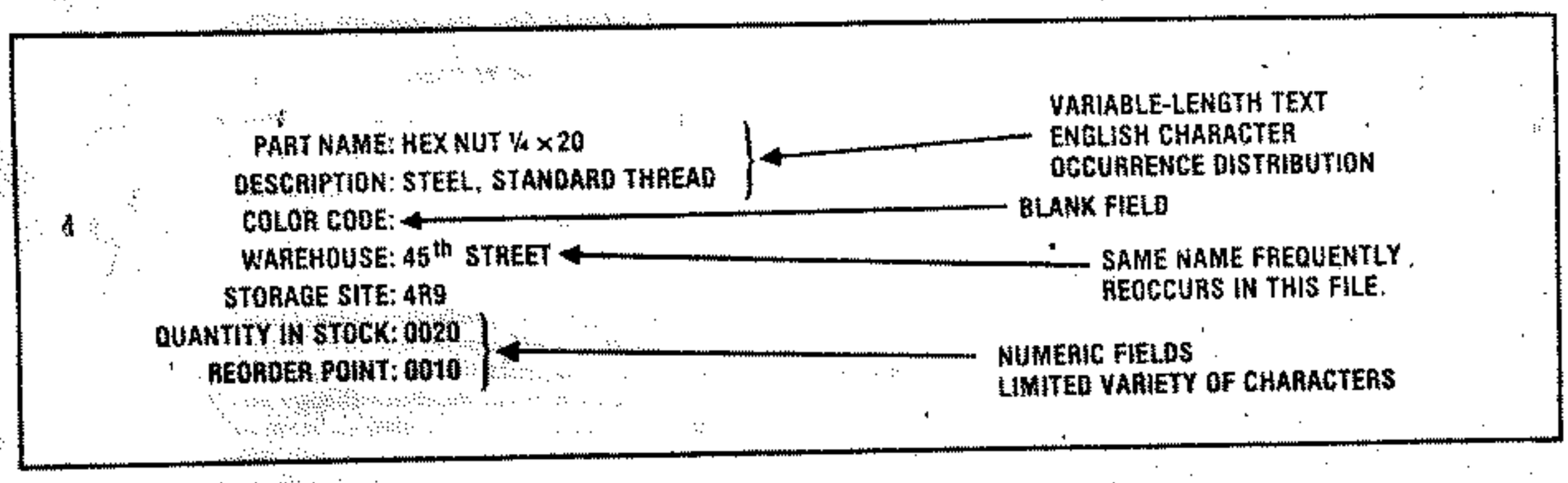


Figure 2. Types of redundant code in a manufacturing parts inventory record.

increasing integration into textual data can pose a compression challenge.

High-usage patterns. Certain sequences of characters will reappear with relatively high frequency and can therefore be represented with relatively fewer bits for a net saving in time/space. In English, a period followed by two spaces is more common than most other three-character combinations and could therefore be recoded to use fewer bits. Many letter pairs, such as ZE, are more common than the individual letter probabilities would imply and might therefore be recoded with fewer bits than the two-character symbols used together. Likewise, unlikely pairs, such as GC, would be encoded with very long bit combinations to achieve better bit utilization. In particular instances of text, certain key words will be used heavily. For example, the word "compression" appears frequently in this article; consequently, if this article were made into a system file, the word "compression" would warrant use of a shorter code. In inventory records, certain identifiers such as warehouse names are extensively reused. Numeric fields contain preferred sequences in the sense that they contain only sequences of digits, with no letters or special symbols intermixed. These could be encoded at less than four bits per digit, rather than the five to eight bits needed for general text.

Positional redundancy. If certain characters appear consistently at a predictable place in each block of data, then they are at least partially redundant. An example of this is a raster-scanned picture containing a vertical line; the vertical line appears as a blip in the same position in each scan, and could be more compactly coded. In inventory files, certain record fields may almost always have the same entry, such as a "special handling" field which almost always has "none" in it. Text, on the other hand, has virtually no positional redundancy in it.

These four types of redundancy overlap to some extent. For example, many inventory numeric fields will contain small integers preponderantly. These could be compressed as a small group of frequently used sequences (the integers), as instances of zero strings in front of the integer values, as a weighting of the character frequency distribution toward zeros, or as a positional bias, in that particular fields will almost always have numeric values with many high-order zeros. The point here is that almost any compression mechanism can exploit that type of redundancy.

The intent here is not to precisely analyze the components of redundancy, but rather to provide a better view of the opportunities and challenges for a compression algorithm.*

*Other types of redundancy exist, beyond those listed here, for example in digitized voice waveforms. There is extensive redundancy in voice, but much of it is best seen in the frequency domain rather than the time domain. Some voice data collected can be eliminated by dropping unnecessary precision in certain contexts, but this technique should be seen as data reduction rather than data compression because it is not reversible. In general, the methods used to compress alphanumeric data are not useful on voice, and vice versa (except perhaps for blank interval encoding).

Methods of compression

The preceding discussion of redundancy types provides a basis for comparing several practical compression methods. More theoretical comparisons are provided by Storer and Szymanki.[4]

Huffman coding. The most popular compression method is to translate fixed-size pieces of input data into variable-length symbols. The standard Huffman encoding procedure prescribes a way to assign codes to input symbols such that each code length in bits is approximately

$$\log_2(\text{symbol probability})$$

where symbol probability is the relative frequency of occurrence of a given symbol (expressed as a probability). For example, if the set of symbols (the input ensemble) is chosen to be the one-byte ASCII characters (a very typical case) and if the blank character is used one-eighth of the time, then the blank character is encoded into a three-bit symbol. If the letter Z is found to occur only 0.1 percent of the time, it is represented by 10 bits.

In normal use, the size of input symbols is limited by the size of the translation table needed for compression. That is, a table is needed that lists each input symbol and its corresponding code. If a symbol is one eight-bit byte (as is common), then a table of 256 entries is sufficient. Such a table is quite economical in storage costs but limits the degree of compression achieved. Single-character encoding can cope with character distribution redundancy, but not the other types. Compression could be improved by using input symbols of two bytes each, but that would require a table of 64K entries at a cost that might not be warranted. Parsing the input data into symbols that are not byte-aligned such as 12 bits, is not likely to improve compression effectiveness and would complicate system design.

A second problem with Huffman encoding is the complexity of the decompression process. The length of each code to be interpreted for decompression is not known until the first few bits are interpreted. The basic method for intepreting each code is to interpret each bit in sequence and choose a translation subtable according to whether the bit is a one or zero. That is, the translation table is essentially a binary tree. Operationally, then, a logic decision is made for each code bit. In working with a disk drive that has a 30M-bps transfer rate, the decompression logic must cycle at that rate or faster to avoid creating a system bottleneck. Decompression at that rate is possible but not simple. In general, decompression with variable-length symbols gives a cost/performance disadvantage whenever the data volume is high.

A third problem with Huffman encoding is that we need to know the frequency distribution for the ensemble of possible input symbols. In the normal case of single-character symbols, character frequency distribution in the data stream is the type of distribution use. Such a distribution is known and is probably reasonably stable for English text. However, the distributions for data files are very application specific and files such as object code, source code, and system tables will have dissimilar characteristic distributions. While it might be possible to have a set of

generic translation tables to cover these various cases, problems arise in having the decompressor use the same set of tables as the compressor. A common solution is to analyze each data block individually to adapt the character distribution uniquely to that block. We must make two passes over the data: (1) a pass to count characters and perform a sort operation on the character table, and (2) a pass for encoding. The derived translation table must be conveyed with the compressed data, which detracts from the compression effectiveness and/or strongly restricts the size of the translation table. This adaptable approach is acceptable if high transfer rates through the compressor are not required and if the data blocks being compressed are very large relative to the size of the translation table.

Run-length encoding. Sequences of identical characters can be encoded as a count field plus an identifier of the repeated character. This approach is effective in graphical images, has virtually no value in text, and has moderate value in data files. The problem with run-length encoding for character sequences intermixed with other data is in distinguishing the count fields from normal characters, which may have the same bit pattern. This problem has several solutions, but each one has disadvantages. For example, a special character might be used to mark each run of characters, which is fine for ASCII text, but not for arbitrary bit patterns such as those in binary integers. Typically, three characters are needed to mark each character run, so this encoding would not be used for runs of three or fewer characters.

Programmed compression. While programmed compression does not meet the transparency constraint desirable for general-use systems, it is discussed here to show the types of techniques used. The programming is generally done by the applications programmer or data management system. In formatted data files, several techniques are used. Unused blank or zero spaces are eliminated by making fields variable in length and using an index structure with pointers to each field position. Predictable field values are compactly encoded by way of a code table—such as when warehouse names are given as integer codes rather than as alphabetic English names. Each field has its own specialized code table that deals with positional redundancy. Since programmed compression cannot effectively handle character distribution redundancy, it is a nice complement to Huffman coding.

Programmed compression has several serious disadvantages. It introduces increased program development expenses; the type of decompression used requires a knowledge of the record structure and the code tables; the choice of field sizes and code tables may complicate or inhibit later changes to the data structure making the software more expensive to maintain. Perhaps most important, programmers will tend to avoid the decompression process because it is relatively slow and, therefore, will work with data in main memory in its compressed format, which confuses and complicates the application program.

Adaptive compression. The Lempel-Ziv[5,6] and related algorithms[7] convert variable-length strings of input symbols into fixed-length (or predictable length) codes. The symbol strings are selected so that all have almost equal probability of occurrence. Consequently, strings of frequently occurring symbols will contain more symbols than a string having infrequent symbols (see example in Table 1). This form of compression is effective at exploiting character frequency redundancy, character repetitions, and high-usage pattern redundancy. However, it is not generally effective on positional redundancy.

This type of algorithm is adaptive in the sense that it starts with an empty table of symbol strings and builds the table during both the compression and decompression processes. These are one-pass procedures that require no prior information about the input data statistics and execute in time proportional to the length of the message. This adaptivity results in poor compression during the initial portion of each message; as a result, the message must be long enough for the procedure to build enough symbol frequency experience to achieve good compression over the full message. On the other hand, most finite implementations of an adaptive algorithm lose the ability to adapt after a certain amount of the message is processed. If the message is not homogeneous and its redundancy characteristics shift during the message, then compression efficiency declines if the message length significantly exceeds the adaptive range of the compression implementation.

Table 1. A compression string table with alphanumeric character strings that are encoded into 12-bit codes. In this example, infrequent letters, such as Z, are assigned individually to a 12-bit code. Frequent symbols, such as space and zero, appear in long strings which in practice can exceed 30 characters in length. Good compression is achieved when a long string is encountered in the input, it is replaced by a 12-bit code, effecting significant space savings.

SYMBOL STRING	CODE
	1
A	2
AB	3
AN	4
AND	5
AD	6
Z	7
D	8
DO	9
DOb	10
b	11
bb	12
bbb	13
bbbb	14
bbbbb	15
bbbbbb	16
0	17
00	18
000	19
0000	20
00001	
	4095
$$$	

b = blank character

The compression procedure discussed here is called the LZW method. A variation on the Lempel-Ziv procedure,[5] it retains the adaptive properties of Lempel-Ziv and achieves about the same compression ratios in normal commercial computer applications. LZW is distinguished by its very simple logic, which yields relatively inexpensive implementations and the potential for very fast execution. A typical LZW implementation operates at under three clock cycles per symbol and achieves acceptably good compression on messages in the magnitude of ten thousand characters in length.

Compression parameters. The preceding discussion of redundancy and compression can be summarized by pointing out the parameters that influence the choice of a compression strategy. The redundancy type found in a certain application is important, but predictability of redundancy type is also important. An adaptive capability in a compression procedure would be of little benefit for applications with predictable redundancy such as in English text, but it would be valuable for business files. System data transfer rates determine if a one-pass procedure is needed for speed, or if the greater overhead of a two-pass approach can be justified by better compression results. The length of the message being transmitted or retrieved has some importance because adaptive techniques are awkward or ineffective on short messages.

It is more difficult to analyze the compression costs that reflect human involvement. Nonadaptive algorithms require prior knowledge of the data characteristics, which are perhaps provided by manual classification of messages into categories having preanalyzed redundancy profiles. Of course, programmed compression offers the greatest density improvement at the cost of very high development and software maintenance costs.

The focus of this article is on the problems found in conventional data storage in commercial computer systems. In that application, the high data transfer rates of magnetic disks preclude extensive runtime overhead, while the diversity of the data stored essentially requires an adaptive procedure. The use of short segments on a disk complicates the use of an adaptive approach, but this is becoming less important as data block sizes on disk are increasing. The LZW method seems to satisfy these requirements better than other compression approaches. Therefore, its characteristics will be used in the following discussion, which points out the opportunities and problems of compression in data storage.

Table 2. Compression results for a variety of data types.

DATA TYPE	COMPRESSION RATIO
English Text	1.8
Cobol Files	2 to 6
Floating Point Arrays	1.0
Formatted Scientific Data	2.1
System Log Data	2.6
Program Source code	2.3
Object Code	1.5

Observed compression results

Effectiveness of compression is expressed as a ratio relating the number of bits needed to express the message before and after compression. The compression ratio used here will be the uncompressed bit count divided by the compressed bit count. The resulting value, usually greater than one, indicates the factor of increased data density achieved by compression. For example, compression that serves to eliminate half the bits of a particular message is presented as achieving a 2.0 compression ratio, indicating that two-to-one compression has been achieved.

Compression ratios, developed by software simulation, are given in Table 2 for several data types. Many of these observations came from compression of backup/recovery tapes from a Sperry Univac 1100/60 machine used in a scientific environment. Most of the samples involved several different files and covered 10^6 to 10^7 bytes each to provide meaningful averages. Several versions of Lempel-Ziv compression algorithms were used; however, since the various algorithms produced results that were consistent to within a few percent, no attempt is made here to distinguish the specific algorithm used for each case.

English text. Text samples for compression were obtained from ASCII word processing files in a technical environment. Results were reasonably consistent for simple text, at a compression ratio of 1.8. Many word processing files, however, compressed better than that when they contained figures, formatted data, or presentation material like viewgraphs. Surprisingly, long individual documents did not compress better than groups of short documents, indicating that the redundancy is not due very much to correlation in content. For comparison, Rubin[2] achieves up to a 2.4 ratio using more complex algorithms. Pechura[3] observed a 1.5 ratio using Huffman encoding. These comparisons are not very reliable, since the subject texts may have dissimilar properties.

Cobol files. A significant number of large Cobol files from several types of applications were compressed, producing widely variable results. Compression depends on record format, homogeneity across data records, and the extent of integer usage. These were eight-bit ASCII files, so the integer data would compress very well. A side experiment showed that one third to two thirds of the space in some of these files appeared as strings of repeated identical characters, indicating a high fraction of blank space. After allowing for that much blank space, there normally was a further two-to-one compression within the useful data. Restricted character usage probably caused two thirds of this latter two-to-one compression, with repeated character patterns giving the rest.

Floating point numbers. Arrays of floating point numbers look pretty much like white noise and so they compress rather poorly. The fractional part is a nearly random bit pattern. The exponent does yield some compression when most of the numbers have the same magnitude. Some floating point arrays expand by 10 percent going through the compression algorithm, when the exponents

vary widely. Expansion under an adverse data pattern can occur under almost any adaptive compression method. Randomized bit patterns usually cause this effect, since there are no redundancy savings to offset the overhead that is inherent in coding the choice of parameter values explicit or implicit in the procedure.

Formatted scientific data. Most data used by Fortran programs tended to compress about 50 percent. This data included input data, primarily integers. It also included print files, which were ASCII coded. Most of these files are short, so our 20-file sample did not encompass a large quantity of data.

System log data. Information describing past system activity, such as job start and stop times, is mostly formatted integers and is therefore reasonably compressible. This log data is used for recovery and constitutes perhaps 10 percent of the data stored on backup/recovery tapes. It tends to be in a tightly packed, fixed-length format, so the compression achieved is due to null fields and repetition in the data values.

Programs. Source code can be compressed by a factor of better than two. It can be compressed better than text because words are frequently repeated and blank spaces are introduced by the source code format. Highly structured programming yields source code with greater compressibility than the average 2.3 factor cited here. Object code, on the other hand, consists of arbitrary bit patterns and does not compress as well. Uneven usage of op codes and incomplete utilization of displacement fields would account for most of the compression achieved. Relocatable code and absolute code differ in compressibility depending on the treatment of blanking for empty variable areas. Actual program files in a development organization were found to be mixtures of source, relocatable, and absolute codes. These mixed files were compressed on the average by a factor of two.

Character size. Sometimes data is recorded in nonstandard character sizes. Our Sperry 1100/60 has a 36-bit word and so, at times, has six-bit, eight-bit, and nine-bit data. The measurements cited in Table 2 were compressed using nine-bit characters throughout. When eight-bit ASCII symbols were being stored in nine-bit bytes, the compression results were adjusted to reflect results as if they had been observed on an eight-bit byte. That is, the observed compression was 9/8 greater than the numbers shown in Table 2. The compressed data stream is virtually the same regardless of whether the source data was in eight-bit or nine-bit symbols, since length of the compressed image depends on the number of symbols encountered rather than their encoding. Six-bit data compressed surprisingly well when compressed on a nine-bit basis, because repeated patterns, such as blank spaces, are word aligned and have the same pattern on each occurrence, even though that pattern is not meaningful when viewed on nine-bit increments. As a side experiment, nine-bit data was compressed as if it were eight-bit characters. Poorer but not unacceptably poor compression resulted. For example, a set of nine-bit files which compressed four-to-one as nine-bit input symbols compressed about 2.8-to-one when processed as a sequence of eight-bit chunks. All of this evidence indicates that compression can be effective even when dealing with mixed character types.

Recovery tapes. Computer backup/recovery tapes (called secure tapes at Sperry) contain copies of active user files. As such, each tape exposes a cross section of data used in its system, although that profile of active files may differ from the profile of all files found on the system disks. Compressing these tapes with the LZW algorithm has produced something less than a two-to-one space reduction in a scientific installation and slightly better than a two-to-one reduction in a program-development-oriented installation. From these results, we might expect a three-to-one reduction in transaction-oriented systems where formatted data predominates.

System considerations

The availability of an appropriate compression method does not assure an effective system. Several problems occur when integrating compression into existing computer systems. These are described here in terms of both problem type and impact on specific computer peripherals.

Unpredictable message length. The length of a compressed image for a given message is unpredictable because it depends on the content of the message. We have no assurance prior to compression that a message will compress at all; in some cases it may even expand a little. Therefore, the space allocated for the compressed image must be at least as big as the space allocated for the original message; this requirement is a complication on allocated devices such as a disk. Further, an update to a data record that alters just a few characters can change the compressed size and may result in a required change in allocation—even for minor update operations. Unpredictability in length also impacts bandwidth requirements; as a result, data path capacities have to be overdesigned to handle peak demands.

Error tolerance. A characteristic of most reversible compression systems is that if one bit is altered in the compressed image, much of the message is garbled. Therefore, a given error rate may be acceptable for the transmission of uncompressed data, but the same rate may be unacceptable for compressed data. Special efforts to detect errors are needed, especially to catch any errors in the compression and decompression processes. Note that a parity bit added to each data character before compression would be removed as redundancy by the compression process, so character parity has no value in checking the compressor. Cyclic checks over the entire message are effective, however.

This intolerance to errors often limits the use of compression. For example, digitized speech can tolerate occasional bit losses, but compression turns these errors into serious losses. On the other hand, since most commercial data storage devices, such as disk and tape, have acceptably low error rates and there is no attempt to use their

data with errors present, compression works well in these circumstances.

Block size. Any compression method that adapts to the statistics of the subject data and has a finite implementation will have compression effectiveness sensitive to message length. Short blocks are penalized by a start-up overhead needed to convey subject statistics. Huffman encoding methods, for example, must send a translation table with the message to give the encoding for each character. Lempel-Ziv methods are inefficient in early sections of the message, until character strings are encountered that repeat earlier strings. A typical LZW implementation uses 12-bit codes with eight-bit input symbols; early inputs cause one code per symbol yielding a 50-percent expansion in the start-up portion of the message.

Large blocks suffer a loss in efficiency because the block lacks stable statistics; this is a typical occurrence in commercial computer data. For example, program development files may contain intermixed blocks of source code, relocatable object code, and load modules of machine code. A translation table built from redundancy statistics on the source code will perform poorly on the object code, which has much different bit patterns. Note that much of the published theory in the compression field assumes arbitrarily large translation tables and ergodic data sources, neither of which occur in practice.

Figure 3 shows typical curves of compression efficiency versus block length for an LZW implementation. Note that compression efficiency is determined by the compressed block size and, as such, is related to the input block size factored by the compression ratio. The curve shown may be moved vertically up or down without loss of meaning; the two-to-one compression ratio curve is shown here for illustration. The diagonal lines indicate fixed input block sizes as they would map into compressed block sizes. Note that data which compresses very well needs to be handled in larger blocks to obtain the highest compression efficiency.

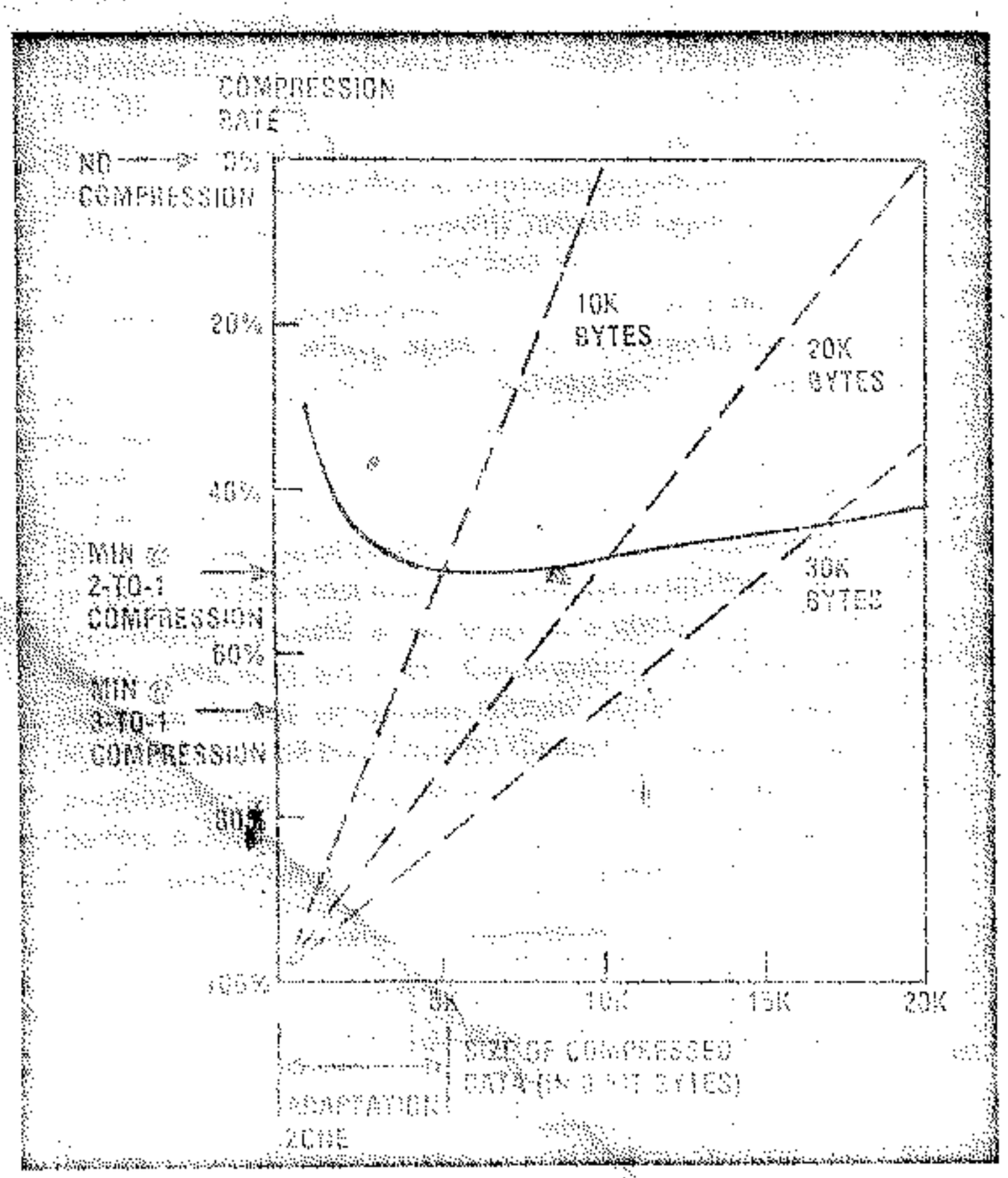


Figure 3. The effect of block size on compression efficiency.

The highest efficiency point shown in Figure 3—about 6K eight-bit bytes after compression—is implementation-dependent and can be adjusted. However, since this curve matches a very convenient implementation, it will be common practice.

System location for compression. When using compression on computer peripherals, a significant system consideration occurs in deciding where to implement compression in the I/O path of Figure 4: in the device controller, channel, or central processor: If the compression operation is executed in the device, it would be transparent to software and its implementation could be adapted to device characteristics. The compression operation inherently requires some buffering because its rate of putting out compressed data varies widely, depending on instantaneous compression ratios. That is, some pieces of a message compress better than others and a highly compressible piece would produce a few output codes in place of numerous input symbols. To reduce the variation in the compressed data rate, a buffer of a few hundred codes would be of value. If implemented in the device controller, this buffering requirement might cause cost problems. Another problem is in channel bandwidth. If data compressible by a factor of three to one is sent to a 3M-bytes per-second device, the channel must deliver 9M bytes per second. This peak rate requirement is variable—depending on compression ratio—so I/O channel sizing can no longer be calculated directly from device transfer rates.

If compression is executed in the channel, it would be visible to operating system software and would require full message buffering in the channel controller (not a traditional design, but a reasonable one). The OS would issue separate operations for compression and I/O transfer, giving it a chance to see the size of compressed images before the data is transferred to a device. Because of data storage requirements, this approach would probably be useful only on selector channels. It does allow the device bus transfer rate to be better utilized because compressed data moves from channel to device, but it also complicates channel and OS designs.

If compression occurs in the central processor, it would be made available to application software and would be a main-memory-to-main-memory operation. This type of compression would improve I/O channel utilization but would increase the main memory cycles required. Invoked optionally by various software packages such as the database manager, this type of compression would not require changes to the OS.

Disk storage. Compression on disk is feasible because the internal data format on disk is generally invisible to

users and can therefore be altered to accommodate compression. The potential for an effectively doubled or tripled data transfer rate on or off disk is very appealing for large computers. Problems can occur, however, with the choice of block size and space allocation procedures.

Traditionally, disks have been used in part to read and write fairly small blocks of characters, such as 80-character records. There are several undesirable ways to do this under a compression systems that involve storing translation tables, but that complicates the system. Fortunately, the present trend in disk subsystem design is to block data to achieve large records on disk and, consequently, better utilize disk bandwidth. This approach requires a buffer memory in the disk controller and often accompanies a caching system for the disk. The use of large blocks introduces some write problems—in read-before-write access-time loss and in recovery provisions—but these drawbacks are present even in caching without compression and so are acceptable.

Traditionally, the OS directly controls space allocation on disk by keeping a map of available space and checking each new write request against the map to determine when and where space is available. If compression is used in a mode transparent to the OS, this direct control is no longer possible. The alternatives are (1) to have the OS be aware of compression and map compressed images into disk space and (2) to move space allocation into the disk controller. Both approaches have disadvantages that will prevent quick incorporation of compression into disk storage.

Tape storage. Tapes seem to be a good prospect for compression because space allocation is not involved (the OS makes no attempt to determine the size of a tape before it is filled) and data block sizes have usually been quite large in practice. Problems occur, however, in read-backwards operations and in intersystem compatibility.

Read backwards, a commonly required capability in a tape system, involves reading data blocks as a character stream in reverse order. Adaptive compression is inherently a one-way procedure, so reversed data blocks must be stored in full before decompression can begin. If compression occurs in the tape controller, buffering must be executed there also, which is not normal in tape subsystems.

Tapes are commonly used for intersystem data exchange, so the use of compression can cause compatibility problems. While most high-performance compression will probably be executed in hardware, it is always possible to decompress with software. Consequently, if occasional tapes are carried to systems without compression hardware, they can be read (slowly) through software decompression. Compression gives all the system appearances of having a higher density tape drive, except that higher media density cannot be reversed using software.

Communications. For years, compression has been commonly used in special communications applications such as facsimile. Adaptive compression has a block size problem in general computer communications, however, because typical interactive use involves messages of a few hundred characters or smaller. Compression looks very attractive for transferring large files. For communications use, software can be ignorant of the existence of compression, so implementing it in the line controller is appropriate.

System-wide application. As the preceding sections imply, the problem with providing compression throughout a computer system is the divergence in methods needed for each device. Communications compression is best implemented in the device controller, tape compression is perhaps best in the I/O channel, and disk compression is best in the processor. Each application involves certain system difficulties that inhibit immediate utilization of compression and the dissimilarity of those difficulties will probably further delay an integrated system-wide compression capability.

LZW compression algorithm

The LZW algorithm is organized around a translation table, referred to here as a string table, that maps strings of input characters into fixed-length codes (see example in Table 1). The use of 12-bit codes is common. The LZW string table has a prefix property in that for every string in the table its prefix string is also in the table. That is if string ωK, composed of some string ω and some single character K, is in the table, then ω is in the table. K is called the extension character on the prefix string ω. The string table in this explanation is initialized to contain all single-character strings.

The LZW string table contains strings that have been encountered previously in the message being compressed. It consists of a running sample of strings in the message, so the available strings reflect the statistics of the message.

LZW uses the "greedy" parsing algorithm, where the input string is examined character-serially in one pass, and the longest recognized input string is parsed off each time. A recognized string is one that exists in the string table. The strings added to the string table are determined by this parsing: Each parsed input string extended by its next input character forms a new string added to the string table. Each such added string is assigned a unique identifier, namely its code value. In precise terms this is the algorithm:

Initialize table to contain single-character strings.
Read first input character → prefix string ω

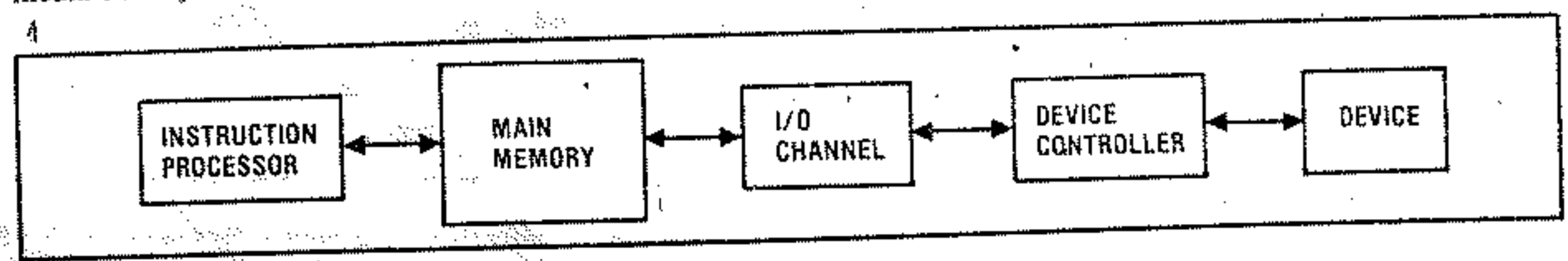


Figure 4. A computer I/O data path from the instruction processor to devices such as tape or disk.

```
Step:  Read next input character K
       If no such K (input exhausted): code (ω) → output; EXIT
       If ωK exists in string table: ωK → ω; repeat Step.
       else ωK not in string table: code (ω) → output;
                                    ωK → string table;
                                    K → ω; repeat Step.
```

At each execution of the basic step an acceptable input string ω has been parsed off. The next character K is read and the extended string ωK is tested to see if it exists in the string table. If it is there, then the extended string becomes the parsed string ω and the step is repeated. If ωK is not in the string table, then it is entered, the code for the successfully parsed string ω is put out as compressed data, the character K becomes the beginning of the next string, and the step is repeated. An example of this procedure is shown in Figure 5. For simplicity a three-character alphabet is used.

Table 3. A string table for the example in Figure 5. The string table is initialized with three code values for the three characters, shown above the dotted line. Code values are assigned in sequence to new strings.

STRING TABLE		ALTERNATE TABLE	
a	1	a	1
b	2	b	2
c	3	c	3
ab	4	1b	4
ba	5	2a	5
abc	6	4c	6
cb	7	3b	7
bab	8	5b	8
baba	9	8a	9
aa	10	1a	10
aa	11	10a	11
aaaa	12	11a	12

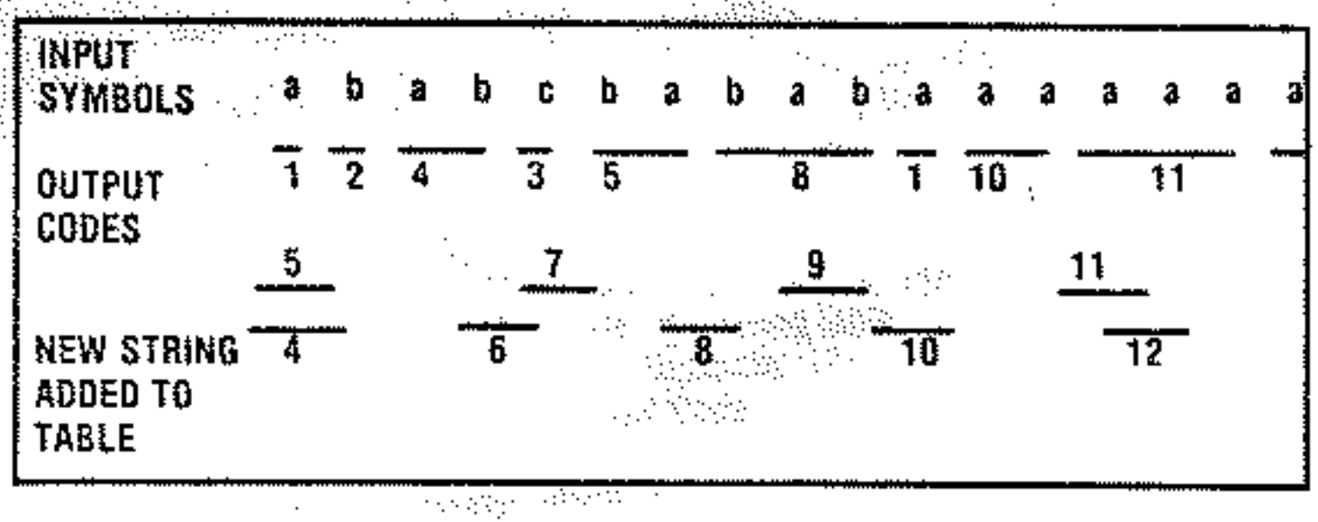


Figure 5. A compression example. The input data, being read from left to right, is examined starting with the first character a. Since no matching string longer than *a* exists in the table, the code 1 is output for this string and the extended string *ab* is put in the table under code 4. Then *b* is used to start the next string. Since its extension *ba* is not in the table, it is put there under code 5, the code for *b* is output, and *a* starts the next string. This process continues straightforwardly.

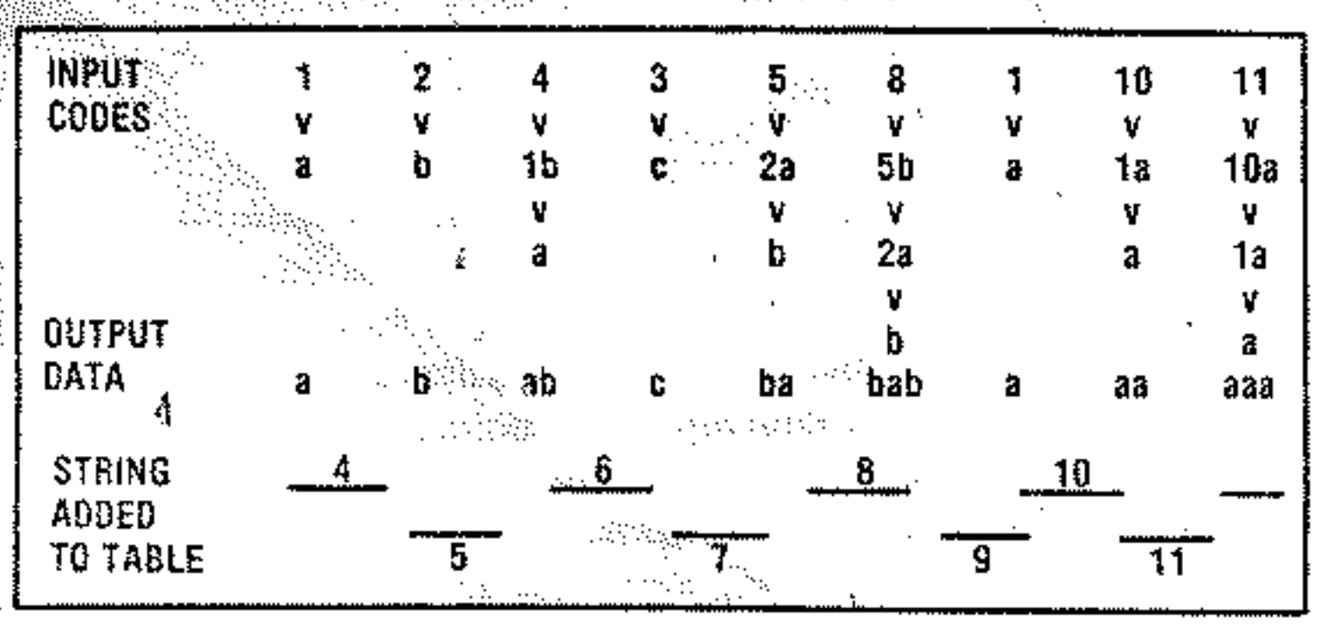


Figure 6. A decompression example. Each code is translated by recursive replacement of the code with a prefix code and extension character from the string table (Table 3). For example, code 5 is replaced by code 2 and *a*, and then code 2 is replaced by *b*.

This algorithm makes no real attempt to optimally select strings for the string table or optimally parse the input data. It produces compression results that, while less than optimum, are effective. Since the algorithm is clearly quite simple, its implementation can be very fast.

The principal concern in implementation is storing the string table. To make it tractable, each string is represented by its prefix string identifier and extension character, so each table entry has fixed length. Table 3 includes this alternative form of the table for the compression example of Figure 5. This form is well suited for hashing methods, and some type of hashing is likely to be used until true associative memories are available in this size.

Decompression. The LZW decompressor logically uses the same string table as the compressor and similarly constructs it as the message is translated. Each received code value is translated by way of the string table into a prefix string and extension character. The extension character is pulled off and the prefix string is decomposed into its prefix and extension. This operation is recursive until the prefix string is a single character, which completes decompression of that code (see Figure 6 for an example). This terminal character, called here the final character, is the left-most character of the string, being the first character encountered by the compressor when the string was parsed out.

An update to the string table is made for each code received (except the first one). When a code has been translated, its final character is used as the extension character, combined with the prior string, to add a new string to the string table. This new string is assigned a unique code value, which is the same code that the compressor assigned to that string. In this way, the decompressor incrementally reconstructs the same string table that the compressor used.

This basic algorithm can be stated as follows:

```
Decompression:  Read first input code → CODE → OLDcode;
                with CODE = code(K), K → output;
Next Code:      Read next input code → CODE → INcode;
                  If no new code: EXIT, else:
Next Symbol:    If CODE = code(ωK): K → output;
                                    code(ω) → CODE;
                                    Repeat Next Symbol
                Else if CODE = code(K): K → output;
                                    OLDcode, K → string
                                      table;
                                    INcode → OLDcode;
                                    Repeat Next Code.
```

Unfortunately, this simple algorithm has two complicating problems. First, it generates the characters within each string in reverse order. Second, it does not work for an abnormal case. String reversal is straightforward and can be done in several ways. For this description, a push-

down LIFO stack for output characters, is assumed, which will be emptied out at the end of each string.

The abnormal case occurs whenever an input character string contains the sequence $K\omega K\omega K$, where $K\omega$ already appears in the compressor string table. The compressor will parse out $K\omega$, send code ($K\omega$), and add $K\omega K$ to its string table. Next it will parse out $K\omega K$ and send the just-generated code ($K\omega K$). The decompressor, on receiving code ($K\omega K$), will not yet have added that code to its string table because it does not yet know an extension character for the prior string. This case is seen in Figure 5 and Table 3 when code 8 is generated; when the decompressor receives code 8 it has only just put code 7 in its string table and code 8 is unknown to it. When an undefined code is encountered, the decompressor can execute translation because it knows this code's string is an extension of the prior string. The character to be put out first would be the extension character of the prior string, which will be the final character of the current string, which was the final character of the prior string, which was the most recently translated character. With these complications remedied, the algorithm is as follows:

```
Decompression:  First input code→CODE→OLDcode;
                with CODE=code(K), K→output;
                                   K→FINchar;
Next Code:      Next input code→CODE→INcode;
                If no new code: EXIT
                If CODE not defined (special case):
                  FINchar→output;
                  OLDcode→CODE;
                  code(OLDcode, FINchar)→INcode;
Next Symbol:    IF CODE=code(ωK): K→stack,
                                  code(ω)→CODE;
                                  Go to Next Symbol;
                If CODE=code(K): K→output;
                                 K→FINchar;
                  Do while stack not empty:
                        Stack top→output; POP stack;
                  OLDcode, K→string table;
                  INcode→OLDcode;
                  Go to Next Code;
```

Note that the string table for decompression is accessed directly by code value, so no hashing is needed. The basic next-symbol step is a simple RAM lookup that produces one output character. In normal implementations, the final single-character code translation does not require a RAM read but is concurrent with a RAM write cycle. As a result, it is reasonably easy to achieve an implementation that consistently produces one output character per clock cycle.

Implementation. The principal implementation decision is choosing the hashing strategy for the compression device. As with any hashing system, the load factor in the table affects search lengths. LZW usually uses 12-bit codes and therefore requires up to 4096 table locations. Using an 8K RAM for the table gives a maximum load factor of 0.5, which produces short average search lengths. Further, a few entries can be lost from the table with only mild degradation of the compression ratio, so codes with long searches can be forgotten. A typical implementation averages about 1.5 RAM accesses per input symbol as a compromise between speed, RAM size, and compression efficiency loss. Each output code also requires about the same time, counting the write cycle for the extended string.

The RAM access time principally determines the clock cycle. Every cycle involves a RAM read (or write) followed by RAM output comparison and the setup of a new address for a new cycle. For high-performance devices a 50-ns RAM might be used, yielding a cycle time of perhaps 65 to 75 ns. Of course, slower logic can be used for lower data rates. The logic structures for decompression are essentially the same as those for compression, so the clock rate will be the same.

Typical hardware structures for compression and decompression are shown in Figures 7 and 8, respectively. The basic flow diagrams are not explained here, but are shown to indicate the general character and extent of the data-path logic. The accompanying control logic is a simple state machine with a half-dozen states in each case. The hardware and software implementations of LZW are Sperry proprietary.

Compression performance depends on the compression ratio of the data being processed. As a rule of thumb, it will take 1.5 cycles per input character and per output code, depending on the hash system used. For eight-bit characters and 12-bit symbols:

$$\text{character rate } (1.5 + (\text{compression ratio})^{-1}) = \text{cycle rate}$$

Consequently, if a clock rate of 15 MHz is used at two-to-one compression, input data can be processed at 7.5M characters per second. Often more critical, however, is the output transfer rate, since this often feeds a real-time device such as a disk.

$$\text{output byte rate } (1 + 1.5(\text{compression ratio})) = \text{cycle rate}$$

To feed a 4M-bytes-per-second disk using a 15-MHz clocked compressor, a compression ratio of only 1.83 can be sustained; any higher compression ratio would starve the output. To feed a 4M-bytes-per-second disk at a compression ratio of three, a 45-ns clock cycle or a hash strategy that gives an average search length of one cycle per character would be required. Since those are tight constraints, this indicates the performance boundaries of the system.

Note that decompression is substantially faster than compression, so its performance is not normally a bottleneck.

Software implementation of LZW is possible but significantly slower. Compression speed is very sensitive to the hash calculation time in the inner loop. Software hashing is relatively slower and less effective than hardware hashing, and generally needs to be hand-coded in machine language. A typical tight coding will average around 40 memory cycles per character and code, so compression speeds of 75K bytes per second are seen on a 1 MIPs machine.

Algorithm variations. Several variations on the basic LZW algorithm just described are possible. The mechanism to reverse output strings could be built with a circular buffer or straight RAM instead of a stack. The latter would require the decompressor string table to contain a length count for each string. Logically, this is easily accomplished but adds complexity.

The compression string table could be initialized to have only the null string. In that case, a special code is necessary for the first use of each single-character string. This approach improves compression if relatively few different symbols occur in each message.

The length of codes used in compressed data can be varied to improve compression early in the message. The first codes contain less information because the string table is relatively empty. With the table initialized to eight-bit characters, the first 256 codes need only nine bits, the next 512 codes need 10 bits, and so on. For example, a ten-thousand character message at two-to-one compression would see a seven-percent improvement in compression ratio using increasing length codes. However, this approach significantly increases logic complexity in both the compressor and decompressor.

The code assignment method could vary from the sequential assignment used in the preceding examples. One interesting alternative is to use the (hash) address of the string location in the compressor's RAM. This saves compressor memory width but requires the decompressor to use the same hashing function to generate code values. Decompression then becomes sensitive to compression implementation techniques, which is a serious disadvantage. This problem is especially hard for software decompression because it is usually difficult to duplicate a good hardware hash function in software.

On the whole, the LZW algorithm as initially described, with 12-bit codes, appears to be a widely suitable system; other variations would appear only in special applications.

The LZW algorithm described here solves several problems that hinder the use of compression in commercial

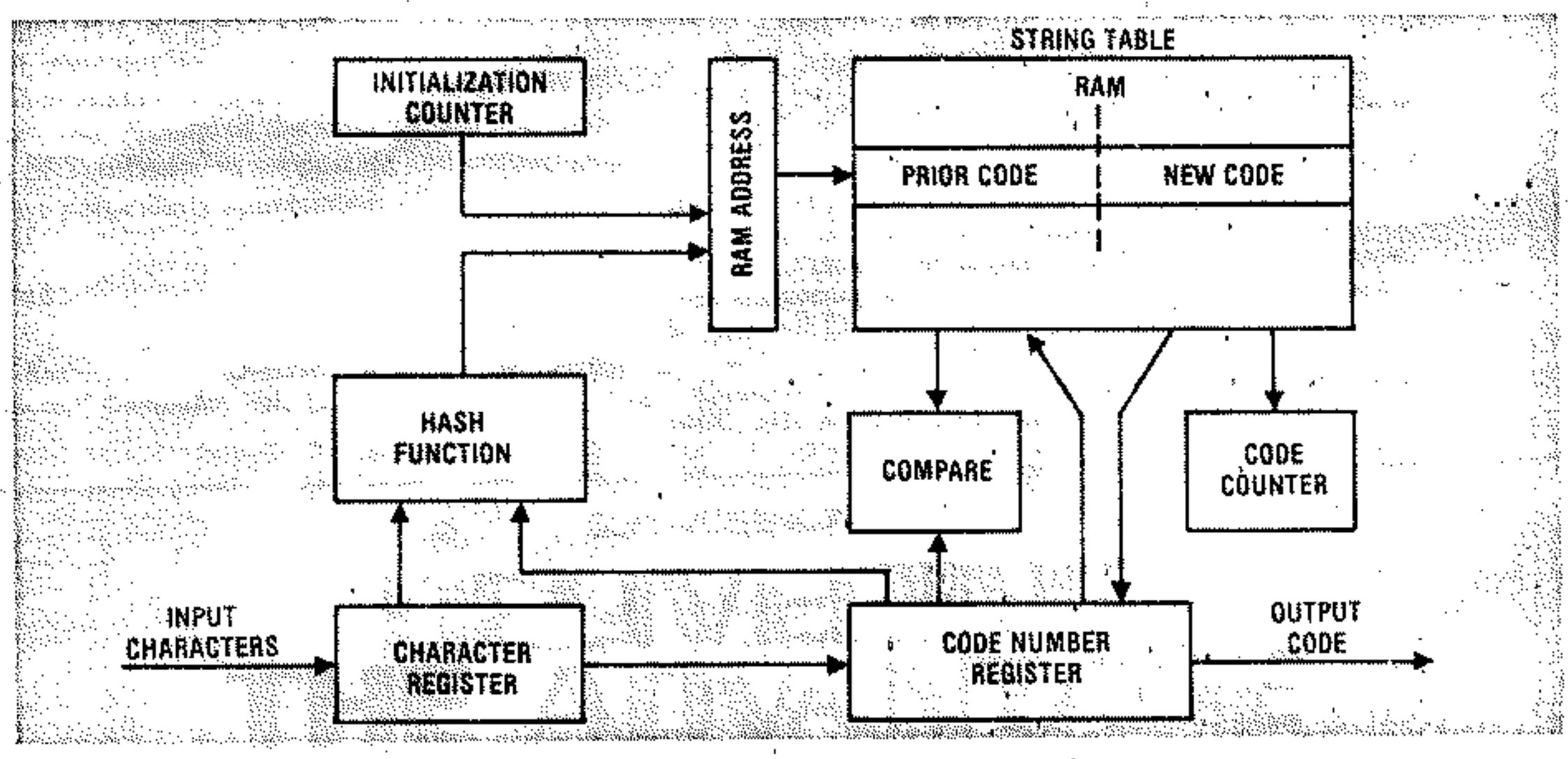


Figure 7. A compressor implementation.

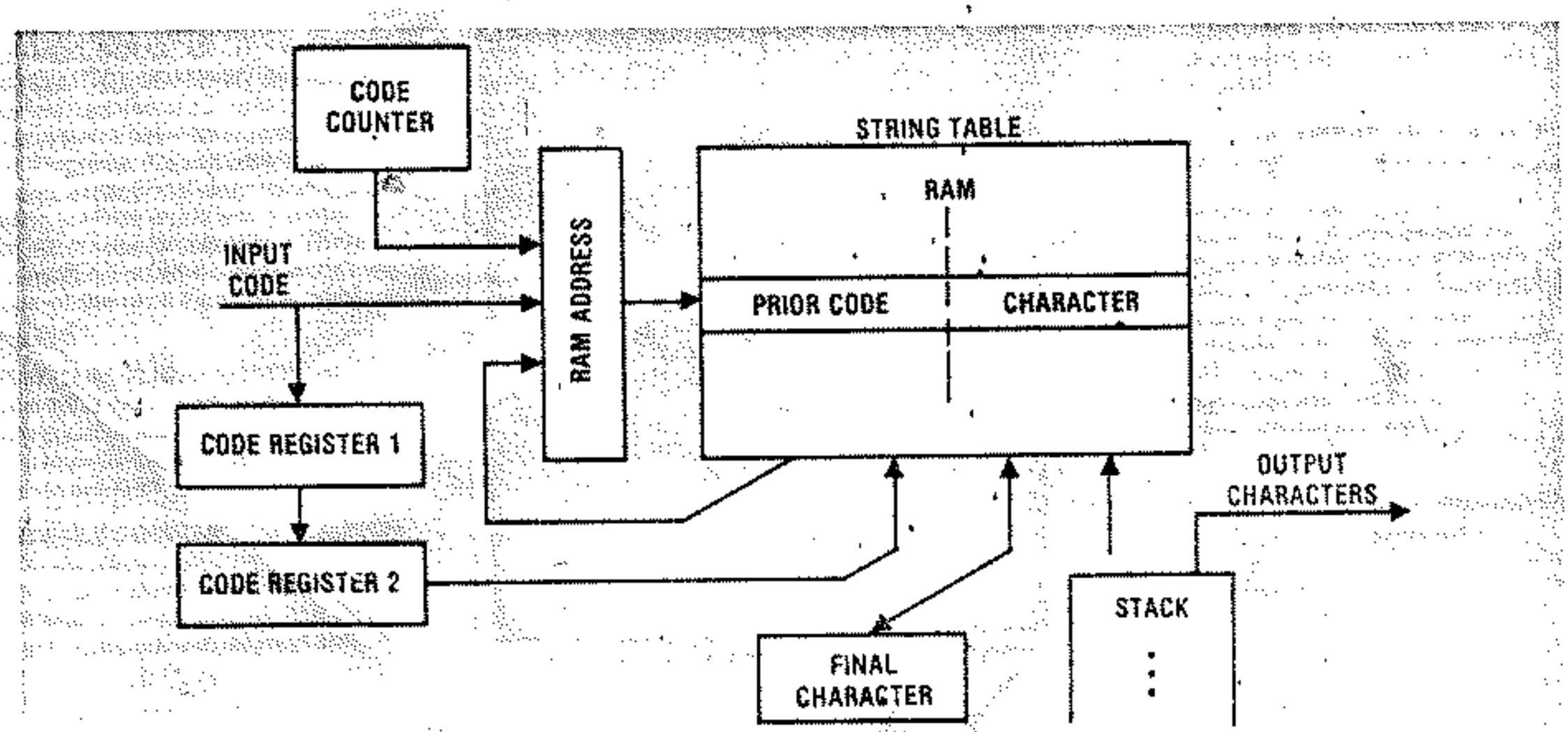


Figure 8. A decompressor implementation.

computer systems. It adapts to the type of data being processed, so it needs no programmer guidance or preanalysis of data. Its simplicity permits very high speed execution, as appropriate to current disk and tape transfer rates. The resulting compression—an expected two-to-three times increase in data density in commercial computer systems—is good enough to be competitive with other methods and, in fact, compares very well in general because it exploits several types of redundancies. Unfortunately, other system-level problems inherent to the use of adaptive compression may prevent its widespread use. These include the unpredictability of compressed image size, the need for character stream buffering, and some constraints on minimum message length. *

Acknowledgments

Several members of the Sperry Research Center have contributed to this data compression effort. Stanley Teeter programmed several versions of the LZW algorithm and collected the data reported in this article. Willard Eastman and Martin Cohn worked on predecessor Lempel-Ziv implementations that illustrated the value of this approach. The efforts of Ted Bonn, in supporting and encouraging this work, are particularly appreciated.

References

1. H. K. Reghbati, "An Overview of Data Compression Techniques," *Computer*, Vol. 14, No. 4, Apr. 1981, pp. 71-76.
2. F. Rubin, "Experiments in Text File Compression," *Comm. ACM*, Vol. 19, No. 11, Nov. 1976, pp. 617-623.
3. M. Pechura, "File Archival Techniques Using Data Compression," *Comm. ACM*, Vol. 25, No. 9, Sept. 1982, pp. 605-609.
4. J. A. Storer and T. G. Szymanki, "Data Compression via Textual Substitution," *J. ACM*, Vol. 29, No. 4, Oct. 1982, pp. 928-951.
5. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Information Theory*, Vol. IT-23, No. 3, May 1977, pp. 337-343.
6. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Information Theory*, Vol. IT-24, No. 5, Sept. 1978, pp. 5306.
7. M. Rodeh, V. R. Pratt, and S. Even, "Linear Algorithm for Data Compression via String Matching," *J. ACM*, Vol. 28, No. 1, Jan. 1981, pp. 16-24.

Terry A. Welch is a senior manager for Digital Equipment Corporation, now on assignment in Austin, Texas, as DEC liaison to MCC's advanced computer architecture program. Prior to joining DEC in 1983, he was manager of computer architecture research at the Sperry Research Center, Sudbury, Massachusetts, for seven years. Previously he taught at the University of Texas at Austin and worked in computer design at Honeywell in Waltham, Massachusetts.

His BS, MS, and PhD degrees were received from MIT in electrical engineering. Welch is a senior member of IEEE and is active in various IEEE-CS activities.

Questions about this article can be directed to the author at MCC, 9430 Research Blvd., Austin, TX 78759.

SOFTWARE ENGINEERS

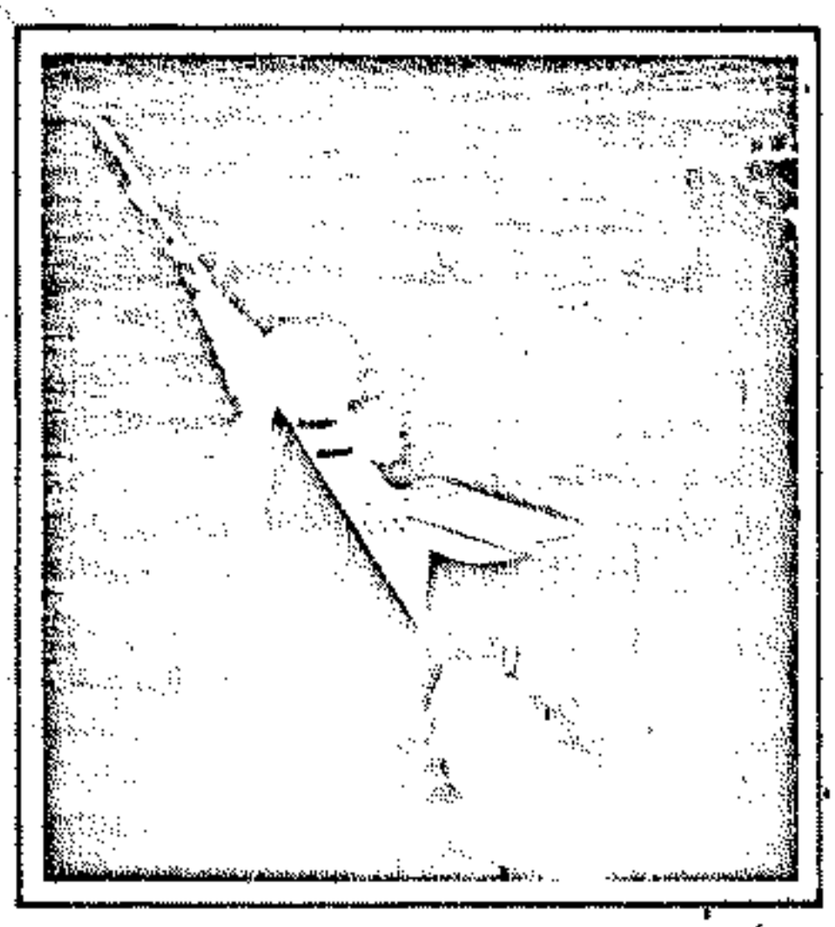
DELIVERING THE ADVANTAGE

At McDonnell Douglas, tough assignments are the rule. And superior careers are the reward.

McDONNELL DOUGLAS ASTRONAUTICS COMPANY represents the highest tradition of success in real-time defense systems development and tactical missile programs. For software engineers, this means exceptional responsibilities in providing the Free World with the vital advantage of superior capability. These career areas reflect an environment where the state-of-the-art is the starting point for challenge.

OPERATIONAL FLIGHT SOFTWARE. Projects focus on embedded missile and satellite applications. A minimum of 5 years of experience with real-time control systems is required.

SHIP MISSILE COMMAND AND LAUNCH SOFTWARE. Responsibilities involve performance analysis and requirements and verification requirements. At least 5 years of experience with displays and operating systems is required. Familiarity with Ada software design methodologies would be helpful.



A BS in electrical engineering or computer science is necessary in either area.

These positions are accompanied by excellent salaries and benefits and the sparkling lifestyle of St. Louis, a city of metropolitan excitement, community spirit and affordable pleasures. If you are seeking the company with an unmatched capacity for accelerating your career, forward your resume and a statement on your professional objectives to:

MCDONNELL DOUGLAS CORPORATION
Attention: Director of Electronics
Post Office Box 516
Department 62-BY-53
St. Louis, Missouri 63166

MCDONNELL DOUGLAS
An Equal Opportunity Employer
U.S. Citizenship Required
CORPORATION