A fast hardware data compression algorithm and some algorithmic extensions

by D. J. Craft

This paper reports on work at IBM's Austin and Burlington laboratories concerning fast hardware implementations of general-purpose lossless data compression algorithms, particularly for use in enhancing the data capacity of computer storage devices or systems, and transmission data rates for networking or telecommunications channels. The distinctions between lossy and lossless compression and static and adaptive compression techniques are first reviewed. Then, two main classes of adaptive Lempel–Ziv algorithm, now known as LZ1 and LZ2, are introduced. An outline of early work comparing these two types of algorithm is presented, together with some fundamental distinctions which led to the choice and development of an IBM variant of the LZ1 algorithm, ALDC, and its implementation in hardware. The encoding format for ALDC is presented, together with details of IBM's current fast hardware CMOS compression engine designs, based on use of a content-addressable memory (CAM) array. Overall compression results are compared for ALDC and a number of other algorithms, using the CALGARY data compression benchmark file corpus. More recently, work using small hardware preprocessors to enhance the compression of ALDC on other types of data has shown promising results. Two such algorithmic extensions, BLDC and cLDC, are presented, with the results obtained on important data types for which significant improvement over ALDC alone is achieved.

Introduction

Several years ago the author began work in Austin to review a variety of data compression algorithms for possible use in improving the capacity of PCMCIA memory cards. These were being considered as removable storage devices for some of IBM's notebook computers, and the initial emphasis was simply to improve capacity for FLASH versions of these cards in particular, as the

An earlier version of this paper was published in the proceedings of a workshop held jointly with the IEEE Data Compression Conference in Snowbird, Utah, March 31, 1995, chairman Robert L. Renner.

0018-8646/98/$5.00 © 1998 IBM

costs of this technology were at that time too high compared to diskettes for such applications.

The scope of the work quickly expanded, however, as it became apparent that data compression technology could have tremendous implications for some major IBM business segments. In particular, its deployment within computer systems to enhance DASD storage capacity, or to increase the effective bandwidth of networking data channels, would present a major competitive advantage.

Lossy compression techniques, often used on image data, achieve better compression by discarding some of the original image (for example, the fine detail information). This is not, however, acceptable for general-purpose use. The data could be financial transactions, accounts, reservation information, executable code and so on, and must therefore be identical to the original when retrieved. Lossless data compression techniques must be used in such situations.

Some compression techniques are static—for example, the CCITT Group 3 algorithm employed in fax data transmission. This works by encoding in a more efficient manner a set of predefined sequences, or strings of data, which are expected to occur frequently. If these do not in fact occur in the data, such algorithms do not achieve compression. Another example would be a text-compression algorithm based on a model of the vocabulary and syntax of the English language. This can achieve good compression on English, but does very poorly on French or German text, executable computer code modules, or other data types.

Static algorithms can be very effective when used in large databases of a relatively uniform data type, for example a mailing list of customer names and addresses. Even if constantly being updated, the character sequences such as "Street," "Avenue," "Jones," or "Smith" are still likely to occur frequently within such a database. For general-purpose data storage or transmission, however, it is not possible to rely on such expectations, and adaptive methods must be used.

Adaptive data compression techniques try to construct models, or look for data sequences derived in some fashion from recent experience. The algorithm thus adapts dynamically to different types of data. There are two classes of adaptive algorithm which are generally acknowledged to be among the most effective, yielding good compression over a wide range of data types. These were both first proposed by A. Lempel and J. Ziv, in 1977 and 1978, and are commonly now referred to as LZ1 and LZ2 respectively [1–3].

Lempel–Ziv algorithms are symbol-based; that is, they operate on data one character (usually a byte) at a time. They achieve compression by locating frequently occurring sequences of such symbols in the input data stream, which are then recoded in more compact fashion. There can be static implementations of LZ1 and LZ2, but they are usually made adaptive. The main distinction between the two classes of algorithm is in the data structure employed and the way references to sequences are coded.

LZ1 algorithms adapt by maintaining a sliding-window history buffer, which can be thought of as a byte-wide shift register. Each incoming data byte is shifted into this history in sequence, the oldest entry being discarded once the history becomes full. LZ2 uses a more complex dictionary structure, which is adapted by adding new sequences to it. The heuristic generally used for this is to append the next incoming data character after using a dictionary entry, to that entry, thus forming a new dictionary entry. Various heuristics are employed once the dictionary is filled: Some software implementations delete the least recently used entry, but the approach usually taken in hardware is to freeze the dictionary, at least while its contents (as measured by the compression achieved) seem to be relevant to the incoming data stream. If compression falls off, such approaches can reset the dictionary and begin to rebuild it, using newer data.

For LZ1, compression is achieved by encoding backward references to the history, specifying a location and a byte count or length. If it is not possible to match the input to a sequence in the history, LZ1 algorithms will output data coded explicitly, usually one byte at a time. Generally, there is a penalty for this: Typically an eight-bit byte is encoded as nine bits. Compression can be obtained once a match of two or more bytes in length is found within the history.

LZ2 algorithms encode a series of dictionary entry numbers, the data and its length both being available from the dictionary entry. When the dictionary is first set up, all possible single-character values are initially included. This ensures that any input data stream can always be encoded. As the dictionary grows, matches are found to entries in the dictionary that have longer sequences. Typically, an LZ2 dictionary may be 4096 entries in size, so a dictionary reference is coded as 12 bits. Compression is then obtained for matching entries two or more bytes in length.

Some results of early initial studies on LZ1 and LZ2 are presented, leading to the conclusion that LZ1 has some distinct advantages, in particular for disk storage use. It not only shows better compression on the smaller data block sizes that are desirable for random-access applications, it is also particularly amenable to a fast and simple CMOS hardware implementation based on the use of a content-addressable memory (CAM) array. This allows the input data-string-matching operations required for LZ1 compression to be performed very efficiently at high speeds, in less silicon area than that of a single chip.

The IBM Microelectronics Division subsequently implemented the CAM-based design approach for one variant of LZ1, called ALDC (adaptive lossless data

compression). The ALDC algorithm is now used in a large number of both IBM and OEM computer storage and telecommunications devices, laser printers, and operating systems. It has been accepted as an ISO and IEC standard, an ECMA standard, an ANSI standard, and a QIC standard.

ALDC compression ratios depend on the data, but are such that capacity for a typical computer DASD storage system can be increased by a factor of 2 to 3 by deployment of this algorithm. Many commercial computer customers often purchase systems for which the DASD cost is the dominant factor, ranging from tens of thousands to millions of dollars per system, so clearly this kind of increase in DASD storage capacity, for less cost than one additional chip, is a significant advance.

More recent work on using small hardware preprocessors to improve the compression of ALDC for other data types has shown promising results. The BLDC algorithm, which combines ALDC with specialized preprocessor hardware for high-resolution binary bitmapped image data, is described. Results are given for one typical application, in this case a set of laser printer page image data files. Compression improvement over ALDC ranges from 1.5 to more than three times better.

Finally, the cLDC algorithm is described, and some application results are given. This algorithm includes a pair of cascaded preprocessors, designed to operate automatically and recode a data stream only when their specific types of data occur. BLDC preprocessing works well on only bitmapped image data, so for other data types the preprocessor must be switched out, or compression will be worse than ALDC. cLDC compression is never worse than ALDC, but can be significantly better on those data types for which the preprocessors operate. One cLDC preprocessor recodes runs of identical data bytes, and this is followed by one designed to recode the Unicode format, an increasingly important text data coding standard used by Java** and other Internet-based applications. ALDC compresses Unicode versions of ASCII text files some 40% worse than the original ASCII, but cLDC is able to remove this penalty completely. In addition, cLDC does almost as well as BLDC on bitmapped image data, so it combines in a single algorithm the capability of ALDC and BLDC with additional improved performance.

Algorithms for DASD or networking applications

Deciding on a suitable lossless algorithm for DASD capacity or network bandwidth enhancement is not simple. From a systems viewpoint, it soon becomes clear that not only are several algorithm characteristics of importance, they tend to vary widely among different candidates. Effective compression clearly matters, but other considerations may in fact sometimes outweigh this. The data used for algorithm evaluation often can markedly influence compression results, and this then leads to the difficult question of what is in fact representative data.

Some of the more important characteristics to be considered must include the compression ratio and its robustness across different types of data, the complexity of the algorithm (whether it is amenable to a hardware or a software implementation), and the resulting costs and speed.

Algorithms, or their implementations, can differ in their symmetry, so that compression versus decompression might be more or less difficult or complex to implement. Some implementations also compromise data compression effectiveness in order to achieve faster speed, and others may exhibit a significant falloff in compression performance if used on smaller amounts or blocks of data. This can be an extremely important characteristic for random-access storage or telecommunications systems applications. Some algorithms achieve good compression but require a two-pass technique. An initial pass over a block of data is used to determine the best algorithmic approach and/or optimal encoding method to be used. The second pass then performs the actual compression operation. However, this is feasible only for buffered applications, tends to be more complex to implement, and cannot achieve the same speeds as a single-pass approach. Utilities such as PKZIP do employ this approach very successfully for archival applications.

A great deal of initial work was done on algorithm comparisons, and it led to three general conclusions (although it is not possible to show all of the results in a paper of this scope). First, the LZ classes of algorithm were confirmed as best for general-purpose use.

Second, it was noted that more complex software LZ2 implementations tended to compress somewhat better than LZ1 on larger amounts of more compressible kinds of data, such as text files, but were less effective on smaller amounts of less compressible data, such as executable code modules.

Third, it became evident that the design compromises necessary to implement the LZ2 algorithm in hardware were in fact affecting data compression performance significantly, to the point that LZ1 was in general superior.

Some results from this early work are depicted in **Table 1**, which compares hardware LZ algorithm implementations available from Stac Electronics (STAC), Advanced Hardware Architectures (AHA), and Infochip Systems Incorporated (ISI). The STAC algorithm is an LZ1 with a history size of 2048 bytes. This is compared with two implementations of LZ2, DCLZ from AHA and a quite similar algorithm from ISI. All three implementations

Table 1 Compression results—hardware LZ1 and LZ2 algorithm implementations.

File data type	*DCLZ* (%)	*ISI* (%)	*STAC* (%)	*LZ2/1*
ISI test data	13.7	13.6	19.3	0.71
Japanese Business	23.1	23.4	19.6	1.19
English Legal	39.8	39.8	33.4	1.19
Lotus 123 work files	45.3	45.0	35.8	1.26
DB4 database	46.5	47.4	35.9	1.31
S/370 object code	39.6	41.7	36.7	1.11
English Technical	41.4	42.5	40.4	1.03
RS/6000 object	65.9	65.5	55.2	1.19
80 × 86 object (MSDOS)	70.2	71.2	57.1	1.24
80 × 86 object (misc.)	71.0	71.9	58.1	1.23
Mandelbrot images	80.9	80.8	73.0	1.11

used comparable chip technology, with similar costs and compression speeds.

The LZ2 algorithms, DCLZ and ISI, were very close in their compression performance, but STAC did significantly better. Results are presented as a percentage of the original size of the given data type. Thus, 100 MB of legal English compresses down to 39.8 MB using either DCLZ or ISI, but will compress to 33.4 MB using STAC. To arrive at an overall figure of merit, given this similarity of DCLZ and ISI, the column LZ2/1 was calculated as [(DCLZ + ISI)/2 × STAC], in effect comparing the average compression achieved by DCLZ and ISI with that achieved using STAC.

Comparisons were done for a very extensive range of data types, and it is clear that the two LZ2 implementations are quite similar, but LZ1 compression is between 3% and 31% better. The one exception to this is unfortunately from a set of test data supplied from one of the two LZ2 manufacturers for marketing purposes. It undoubtedly shows that some kinds of data compress better using LZ2 than LZ1, but in our experience it is quite anomalous, and is probably atypical.

Other fundamental differences between LZ1 and LZ2 became evident from this early work. More complex LZ2 algorithms achieved up to 20% better compression than LZ1 on large blocks of data, but simpler LZ2 implementations, using the dictionary-freeze heuristic, could be up to 40% worse on data blocks of the smaller sizes desirable for DASD storage and communications systems applications.

LZ1 also tends to be highly asymmetric, in that decompression is very easy and fast in software or hardware, compared to compression. Most computer systems typically read DASD data about four times more often than writing, so this is an advantage. With LZ2, both data compression and decompression can be much more complex, depending on the dictionary-management heuristic employed.

Finally, during the course of this work, the LZ1 approach was found to lend itself especially well to extremely fast and efficient hardware implementations involving the use of a CAM to store the history. This was particularly attractive for use in high-performance tape and disk storage applications, and led directly to the development of the ALDC variant of LZ1.

CAM-based LZ1 compression hardware

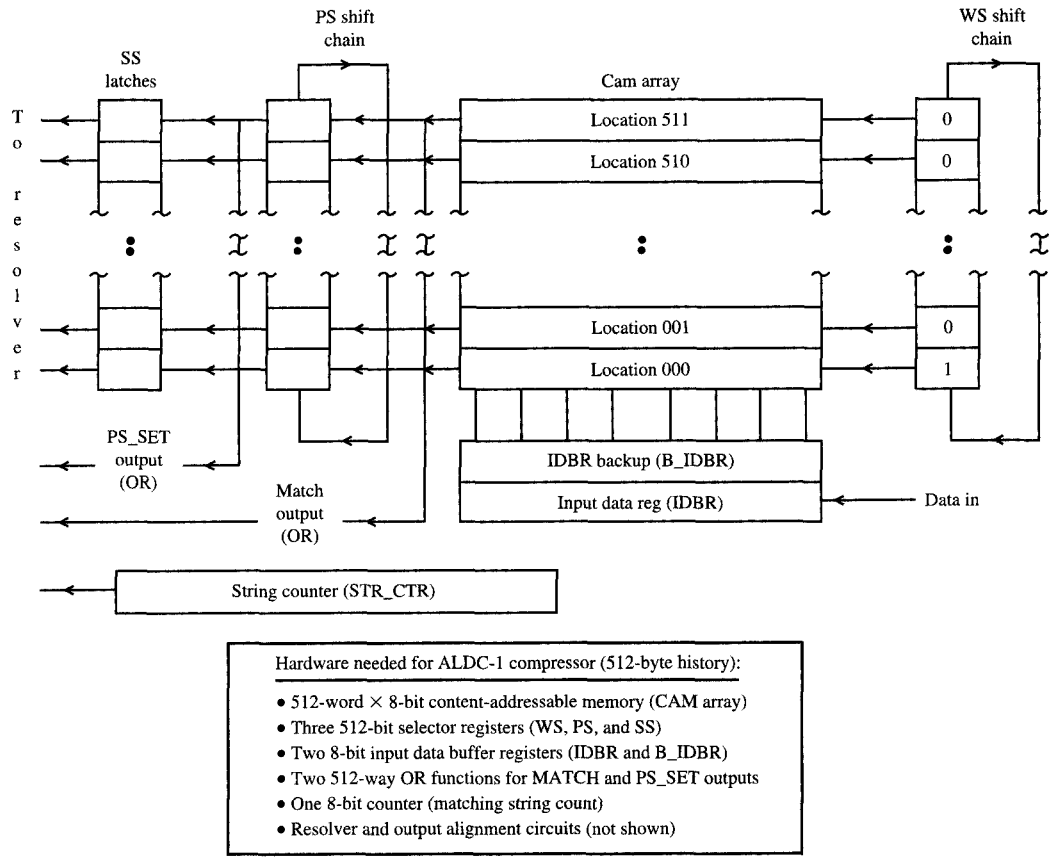
There are two fundamental requirements for implementation of the LZ1 data compression algorithm. First, each byte of input data processed must be entered in sequence into a history structure, the oldest one being discarded once this history becomes full. Second, to achieve compression, the incoming data stream must be compared with the current history content to determine whether it matches a sequence, or string of bytes, which occurred recently and is thus still within the history.

Figure 1 shows some of the internal dataflow of the CAM-based ALDC compressor. The CAM's parallel processing capability not only allows extremely fast compression; it also permits what we call an exhaustive search. Incoming data bytes are compared to all possible candidates in the history in one cycle. Software LZ1 algorithms must limit the search processing in order to achieve an acceptable speed. This would be called a partial search, since all possibilities are not in fact considered. As a consequence, ALDC is able to achieve comparable data compression using smaller history sizes. This generally will also yield better compression on smaller data block sizes, since such a history is filled, and thus fully effective, at an earlier point in processing the block.

The design operates by performing what we call a SEARCH_WRITE in one associative cycle of the CAM, for each incoming data byte. The current history is held in the CAM, and to start a string-matching operation, the SEARCH part of an initial SEARCH_WRITE cycle is used to SET all bits in the PS register corresponding to CAM locations where a match with the incoming data byte is found.

The WS register has only a single bit set, and this determines the CAM location to which the incoming data byte value is stored during the WRITE part of the cycle. Once the cycle is complete, the WS and PS registers are shifted by one bit position, ready to process the next byte of data.

The WRITE part of the cycle essentially does the required LZ1 history update function by storing each incoming byte processed, in sequence, to consecutive CAM locations. The single bit in the WS register wraps around to the first location after the last one is reached, and so this design uses what we call a circular history,



Internal dataflow outline—CAM-based ALDC compressor.

rather than the classical linear or sliding-window history structure, as originally proposed for LZ1. The two are equivalent in function, however.

Once a match is started, the SEARCH portions of subsequent SEARCH_WRITE cycles are used only to RESET the PS register bits, adjacent to history locations where there is a mismatch. As long as at least one PS bit is still left SET, the string-matching process continues, wrapping around the history indefinitely, unless an upper limit to the matching string length set by the length count encoding format is reached. A counter is incremented each SEARCH_WRITE cycle to track the length of the matching data string, and the PS content is loaded into the SS register before each shift of the PS and WS registers. When an incoming data byte occurs which breaks a string match, the SEARCH_WRITE cycle for that data byte will result in all PS bits being RESET. This condition is detected by the PS_SET output OR gate, shown in Figure 1. The total length for this matching string is then available from the counter STR_CTR, while the END position(s) of string(s) are marked by one or more bits in register SS. One of these ending points is selected by the resolver circuitry (not shown), and the start point is computed by an addition of the STR_CTR value (modulo the history size) to its address value. A COPY_POINTER is then encoded and output for this string, and a new string-matching operation is begun.

A more detailed operational description can be found in [4]. Designs of this kind can process one input byte for each SEARCH_WRITE cycle of the CAM, and are capable of extremely fast compression speeds. IBM's production CMOS 5 processes now easily yield a SEARCH_WRITE cycle time for the CAM of less than 10 ns, so sustained input data rates of 100 MB/s are readily achieved. At compression ratios of 2× to 3×, average output data rates thus range from 33 to 50 MB/s and are well matched to the raw data speeds of high-performance DASD and archival tape storage devices.

<Compressed_Data> := [0 <LITERAL> | 1 <COPY_PTR>] 1 <CONTROL>
<LITERAL> := <b><b><b><b><b><b><b><b> (8-bit byte data)
<b> := 0 | 1 (the | symbol denotes OR)

<COPY_PTR> := <length_code> <displacement>
<length_code> := (the length coding can be 2, 4, 6, 8, or 12 bits)

(length_code)	(field value)	(COPY_PTR length)
00	(0)	(2 bytes)
01	(1)	(3 bytes)
10 00	(2)	(4 bytes)
: : : :	: : :	: : : : : :
10 11	(5)	(7 bytes)
110 000	(6)	(8 bytes)
: : : : : :	: : :	: : : : : :
110 111	(13)	(15 bytes)
1110 0000	(14)	(16 bytes)
: : : : : : : :	: : :	: : : : : :
1110 1111	(29)	(31 bytes)
1111 0000 0000	(30)	(32 bytes)
: : : : : : : : : : : :	: : :	: : : : : :
1111 1110 1110	(268)	(270 bytes)
1111 1110 1111	(269)	(271 bytes)

<displacement> := <b><b><b><b><b><b><b><b><b> (9-bit)

<CONTROL> :=

(ctl_code)	(field value)	(control specified)
1111 1111 0000	(270)	(the 12-bit field values of 270 to 284 are reserved codes and cannot be used)
: : : : : : : : : : : :	: : :	
1111 1111 1110	(284)	
1111 1111 1111	(285)	(End_Marker control)

(a)

(identical to ALDC_1 definition except for 10-bit displacement field)
<displacement> := <b><b><b><b><b><b><b><b><b><b> (10-bit)

(b)

(identical to ALDC_1 definition except for 11-bit displacement field)
<displacement> := <b><b><b><b><b><b><b><b><b><b><b> (11-bit)

(c)

Figure 2

Formal definitions of ALDC algorithms: (a) ALDC-1 (512-byte history); (b) ALDC-2 (1024-byte history); (c) ALDC-4 (2048-byte history).

LZ1 decompression is relatively simple, requiring only a byte-wide SRAM of the same size as the history. An address counter is used, in much the same way as the WS pointer in the CAM, to store each output data byte sequentially into this SRAM. This satisfies the history update requirement. Decoding of COPY_POINTERS simply requires that the specified string be copied out of the SRAM once its start address and the byte count have been decoded. Output of each decompressed data byte thus requires one READ_WRITE cycle of the SRAM, analogous to the CAM's SEARCH_WRITE cycle during compression. In the READ phase, the data byte is fetched from its location in the history. It is then both output and copied to the current update location in this same history SRAM during the WRITE phase. Control is relatively trivial, and one small counter with two SRAM address register/counters (one for READ, one for WRITE), is all that is required. Decompression speeds can be made faster than compression speeds in the same technology, if required, but there has been no demand for this as yet.

The initial IBM ALDC chips used entirely separate decompression and compression engines [5] and were in fact configured so that both could be operated simultaneously. However, in our later designs, we chose to add a conventional address decoder to the CAM so that it can also function as an SRAM during an LZ1 decode function. This results in a very compact hardware encoder/decoder, which we call a CRAM design, as it combines a compression engine CAM and a decompression engine RAM into one silicon array. The CRAM designs are as fast as their predecessors, but cannot offer simultaneous compression/decompression on separate data streams.

The CRAM-based family of ALDC lossless high-speed hardware compression designs are available either as separate chips, or preferably as ASIC cores, for integration on a single chip with other functions. These are the smallest and fastest solutions in the industry, offering speeds up to 100 MB/s and requiring a chip area of less than 5 mm^2 using IBM's current production CMOS process. For the few customers requiring simultaneous compression/decompression capability, we can therefore put two such CRAM engines on the same chip, and the silicon area needed is still modest.

ALDC encoding structure

LZ1 algorithms encode their compressed output as a mixture of what are called LITERALS and COPY_POINTERS. COPY_POINTERS include two components, a byte count and a displacement. The latter indicates to a decompressor the start location within its history from which it must begin copying the matching data string. LITERALS usually encode a single byte of data which did not form part of a matching string. The COPY_POINTERS provide compression if the length and displacement components can be encoded in fewer bits of information than the specified data byte string.

An LZ1 decompressor builds and maintains an identical history copy, which it updates in the same manner as the encoder, as each LITERAL or COPY_POINTER is processed. Both histories are initially set empty at the start of an operation. The encoder history becomes different while a pointer is being generated, but once

the decoder has processed this pointer, the histories are identical once more.

The CAM-based designs can be used with any LZ1 encoding scheme, but always process all possible matching data strings to find the longest, and this has implications for optimal COPY_POINTER encoding. Extensive tests showed that the displacement components had a tendency to be distributed uniformly over the history address space, so a flat binary field, base-2 log of the history size in length, is used in ALDC to give optimal encoding for the displacement. The length component values were found to be very similar to classical LZ1 distributions, highly skewed toward the low end of the range, but with some incidence of higher values on much of our internal IBM data.

There are two reasons for this: Many commercial applications tend to fill unused fields with blanks or leading-zero values; also, assemblers and compilers often load all arrays, variables, and other storage areas with zeros for consistency in case the areas are referenced before being properly initialized.

Therefore, we elected to use a variable code for the length component encoding, as published by Brent [6] and others. A variety of encoding schemes were tried on our data, and a logarithmic code (suggested by E. Karnin of IBM Haifa in a private communication) was finally used in modified form. This was limited to a maximum-length-count component of 271 bytes, with some spare codes reserved for future architectural use.

The scheme originally proposed by Storer and Szymanski [7], then implemented as LZSS and published by Bell [8], provides an effective way to differentiate between LITERAL and COPY_POINTERS, and is also used in the ALDC coding scheme.

Figure 2 details the ALDC encoding format for the three defined history sizes of 512, 1024, and 2048 bytes, which we refer to as ALDC_1, ALDC_2, and ALDC_4, respectively. This is because the CAM-based designs all use a 512-byte CAM history array macro, and one, two, or four of these are laid down on the chip as needed.

ALDC_1 was the initial preferred history size, since the CAM occupies most of the design area and we found that doubling history size roughly doubles cost and power, but provides a much smaller increase in data compression ratio. As the density of our CMOS processes has increased, it is now likely that ALDC_2 will replace ALDC_1 as a preferred history size.

The ALDC algorithm is now the QIC-154 tape compression standard for the quarter-inch cartridge tape drive industry. Also, it has been accepted as the ECMA-222, ISO/IEC 15200, and ANSI x3.280-1996 data compression standards. References [9] and [10] contain additional descriptive material on ALDC.

Table 2 Overall compression results on file CALGARY_CORPUS, various algorithms.

Algorithm	*Compressed results* (bytes)
SXTERSE	1,256,864 (2.59)
COMPRESS	1,361,713 (2.39)
ALDC_4	1,442,563 (2.25)
ALDC_2	1,538,420 (2.11)
ALDC_1	1,648,880 (1.97)
STAC_F	1,798,568 (1.81)
STAC_H	1,500,746 (2.17)
IIT_F	1,744,897 (1.86)
IIT_H	1,507,689 (2.16)
DCLZ	1,638,296 (1.99)
ISI	1,562,716 (2.08)
IDRC	2,005,340 (1.62)

Comparative results on the CALGARY corpus

Table 2 compares results obtained using some twelve algorithm variants to compress the entire CALGARY corpus of test data. The CALGARY corpus is a standard set of mostly textual data of various kinds, but it also contains some source code, executable code, an image data example, and some geodesic data [11]. The original size of the CALGARY_CORPUS file is 3251493 bytes; it is simply the corpus files concatenated in the following order: BIB, BOOK1, BOOK2, GEO, NEWS, OBJ1, OBJ2, PAPER1, PAPER2, PAPER3, PAPER4, PAPER5, PAPER6, PIC, PROGC, PROGL, PROGP, and TRANS.

Software LZ1 algorithms from Integrated Information Technology (IIT) and STAC Electronics (STAC), used in some of their PC disk-doubler products, are available in FAST and HI compression versions. These are distinguished by use of the _F and _H suffixes, respectively. Each of these algorithm variants uses a 2048-byte history size.

The LZ2 algorithms are represented by SXTERSE, a complex IBM internal software implementation; DCLZ, a hardware algorithm implementation from Advanced

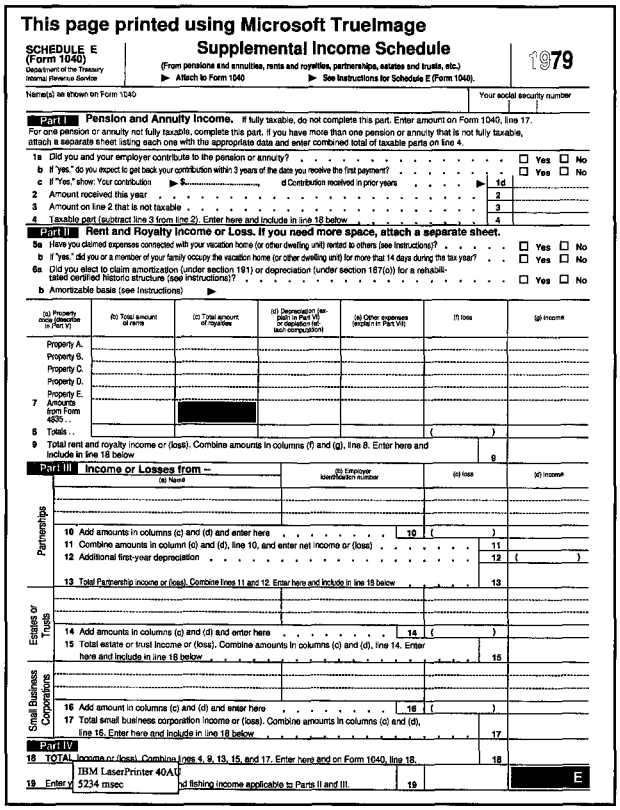


Figure 3

Full-page bitmap (FPBM) image of a U.S. Internal Revenue Service form — file 1040FPBM.

Hardware Architectures (AHA); a very similar algorithm from Infochip Systems Incorporated (ISI); and COMPRESS, a standard utility available within UNIX.**

IDRC, an IBM arithmetic coder hardware algorithm used on the 3480 and 3490 mainframe tape drives, is also included. The overall compression ratios are shown in brackets under the output file sizes in bytes.

Notice that ALDC_4, at the same history size of 2048 bytes, compresses better than either STAC_H or IIT_H, but more significantly, compression for ALDC does not fall off very much at the smaller history sizes. The faster software implementations, STAC_F and IIT_F, however, pay greater penalties in compression ratio for their increased speed, and ALDC_1 still shows better compression on only a 512-byte history size. This is primarily because the faster software implementations do not pursue an exhaustive search strategy over all possible matching strings within their history structures, so as to reduce the processing requirements and hence improve compression speed.

BLDC extension of ALDC for bitmap image data

Small laser printers are a common output device for business and other personal computer system applications, and are thus manufactured in large volumes. It is necessary that an image to be printed be available in bitmap form within the printer before starting to move a sheet of paper through the print path, as the process requires constant speed to achieve acceptable print quality. At high resolution, a significant amount of DRAM is required within such printers for page image storage, and microcoded compression algorithms are used to reduce the cost. The performance of these becomes a serious limitation as image resolutions and print speeds increase.

Conventional LZ1 algorithms do not perform especially well on data of this type, but by adding a small hardware preprocessor to recode the bitmap data stream before feeding to an ALDC compressor, a much better overall result is achieved.

The method adopted is simply to encode consecutive runs of identical data bits as a single-byte-count field,

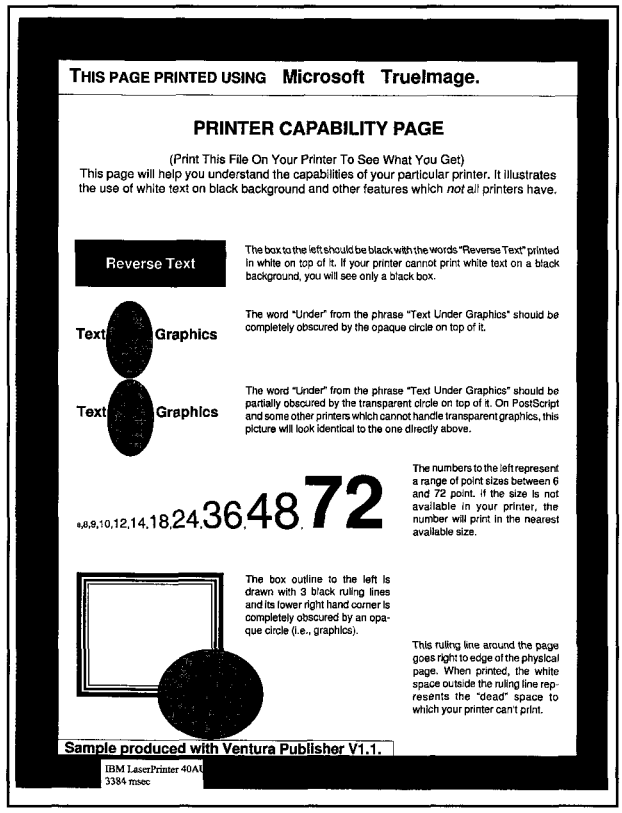


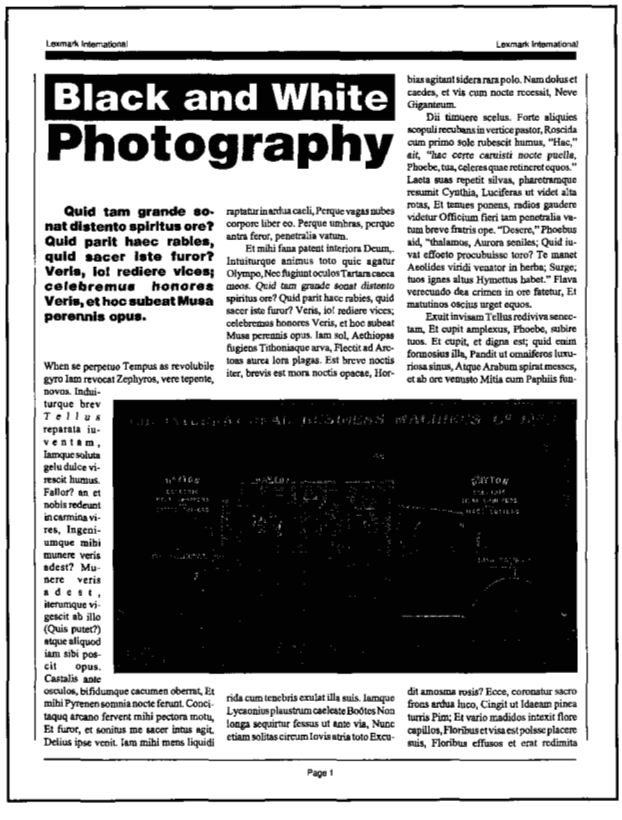
Figure 4

FPBM image of a laser printer test page — file ABILFPBM.

assuming that the data begin with at least a single 0 bit. Provided this is so, and runs less than a length of 254 occur, the encoded bytes are simply the run-length values for 0s and 1s alternately. The value 255 is an extended count code, indicating that the additional length of this run over 254 bits is continued on the next byte, encoded in the same fashion. Byte value 00 denotes a run length of zero; it is used if the data do not begin with a 0 bit, with the next byte value describing the length of the first run of 1s.

The code byte-stream output from this preprocessor is then fed into a standard ALDC_1 encoder. For decompression, an ALDC decoder generates code byte values which are then fed into a hardware postprocessor to reconstruct the original data bit stream.

The hardware circuitry for this additional processing function requires an increase in silicon area that is quite small compared to that for ALDC, and it is able to operate at comparable speed. This combined algorithm [12] is termed bitmapped lossless data compression, or BLDC. Other types of preprocessor could be devised for other image data applications, for example gray scale or color, and lossless or lossy compression.


Lexmark International

Lexmark International

Black and White Photography

Quid tam grande sonat distento spiritus ore? Quid parit haec rabies, quid sacer iste furor? Veris, io! rediere vices; *celebremus honores* Veris, et hoc subeat Musa perennis opus.

When se perpetuo Tempus as revolubile gyro Iam revocat Zephyros, vere tepente, novos. Induturque brev *T e l l u s* reparata *u v e n t a m*, Iamque soluta gelu dulce virescit humus. Fallor? an et nobis redeunt in carmina vires, Ingeniumque mihi munere veris adest? Munere veris *a d e s t*, iterumque virescit ab illo (Quis putet?) atque aliquod iam sibi poscit opus. Castalis ante oculos, bifidumque cacumen oberrat, Et mihi Pyrenen somnia nocte ferunt. Concitaquq arcano fervent mihi pectora motu, Et furor, et sonitus me sacer intus agit. Deliius ipse venit. Iam mihi mens liquidi raptatur in ardua caeli, Perque vagas nubes corpore liber eo. Perque umbras, perque antra feror, penetralia vatum.

Et mihi fana patent interiora Deum,. Intuiturque animus toto quic agatur Olympo, Nec fugiunt oculos Tartara caeca meos. Quid *tam grande sonat distento* spiritus ore? Quid parit haec rabies, quid sacer iste furor? Veris, io! rediere vices; celebremus honores Veris, et hoc subeat Musa perennis opus. Iam sol, Aethiopas fugiens Tithoniaque arva, Flectit ad Aretoas aurea lora plagas. Est breve noctis iter, brevis est mora noctis opacae, Horrida cum tenebris exulat illa suis. Iamque Lycaonius plaustrum caeleste Bootes Non longa sequitur fessus ut ante via, Nunc etiam solitas circum Iovis stria toto Excudit amosma rosis? Ecce, coronatur sacro frons ardua loco, Cingit ut Idaeam pinea turris Pim; Et vario madidos intexit flore capillos, Floribus et visa est polsse placere suis, Floribus effusos et erat redimita bias agitant sidera rara polo. Nam dolus et caedes, et vis cum nocte recessit, Neve Gigantem.

Dii tuuuere scelus. Forte aliquies scopuli recubans in vertice pastor, Roscida cum primo sole rubescit humus, "Hac," ait, "hac certe caruisti nocte puella, Phoebe, tua, celeres quae retinerat equos." Laeta suas repetit silvas, pharetramque resumit Cynthia, Luciferas ut videt alta rotas, Et tenues ponens, radios gaudere videtur Officium fieri iam penetralia vatum breve fistris ope. "Desere," Phoebus aid, "thalamos, Aurora seniles; Quid iuvat effoeto procubuisse toro? Te manet Aeolides viridi venator in herba; Surge; tuos ignes altus Hymettus habet." Flava verecundo dea crimen in ore fatetur, Et matutinos oscius urget equos.

Exuit invisam Tellus redivlva senectam, Et cupit amplexus, Phoebe, subire tuos. Et cupit, et digna est; quid enim formosius illa, Pandit ut omniferos luxuriosa sinus, Atque Arabum spirat messes, et ab ore vemusto Mitia cum Paphiis funde

Page 1

FPBM image of a laser-printed black-and-white photograph combined with text from a personal publishing application — file BWP1FPBM.

FPBM image of a computer-generated graphic image — file GOLFFPBM.

Figures 3–7 show some example laser printer full-page bitmap (FPBM) test images. The 1040FPBM form is very typical, since many software applications for tax preparation, mortgage loan processing, insurance loss or medical claims, and so on use such printers to output similar forms, with the individual data included.

One example each for most of the other image types is shown, with the names of similar image files (not shown) following in parentheses. ABILFPBM (FONTFPBM and SPRLFPBM are similar) is a printer capability page. BWP1FPBM (BWP2FPBM, BWP3FPBM, and BWP4FPBM are similar) shows a personal publishing page, with text plus a halftone image. GOLFFPBM (MEOWFPBM is similar) is a typical business image graphic. The SCOPFPBM example (STARFPBM is similar) is another personal publishing page, but with text and line drawings. File MOREFPBM (not shown) is a typical business invoice/order form.

Ventura Scoop

SPECIAL EDITION — APTOS POST TYPOGRAPHY — MARCH, 1987

Xerox Shows Off Ventura Publisher at Conference

BEVERLY HILLS (VP) — Xerox Corporation has introduced version 1.1 of its first electronic publishing software product that runs on industry standard personal computers. Xerox chose the Seybold Conference to announce the price and availability of the new revision to the industry standard software package. Conference attendees were impressed by the eighty-one new features, all of which were added without compromising the speed of the product and its depth of functionality.

Product now widely available.

The Xerox Desktop Publishing Software Series: Ventura Publisher Edition is available through Xerox authorized dealers (including ComputerLand, Microage, and Pactel), and the Xerox Business Software Center via (800)-822-8221, and the Xerox general line sales force. Commented one observer, "This breadth of distribution represents Xerox's commitment to the mainstream of the PC-based market."

Ventura Publisher Edition allows personal computer users to merge text and graphics to create publishing-quality documents, such as newsletters, technical manuals, books, bids and proposals, that might otherwise be sent to a print shop or typesetter. The package runs on the Xerox 6065, IBM PC/XT, IBM PC/AT, and other PC compatibles. It supports popular laser printers, including the Xerox 4045, the Apple LaserWriter and the HP Laser Jet.

Pioneers in the field

"As one of the pioneers in the field of electronic publishing, Xerox fully understands users' requirements for a desktop publishing software product," said James N. Brown, vice president, office systems

IBM LaserPrinter 40AU
16383 msec

Shuttle

Version 1.1 Redefines Desktop Publishing—Again

MORGAN HILL (VP) — Xerox Ventura Publisher Edition version 1.1 has added new meaning to the term "Desktop Publishing." Before the introduction of Ventura Publisher Edition, desktop publishing referred primarily to advanced drawing packages that were extended to handle different text fonts.

These types of packages were characterized by a hand-intensive approach that attempted to mimic what graphic artists and typesetters were used to doing using the personal computer screen as an electronic paste-up board. While this approach was easy for artists to pick up, it did not result in much time-saving because the user was still faced with the drudgery of hand-adjusting each piece of text on the page.

Fortunately, the software developers at Ventura Software Inc. recognized this and adopted a **style sheet** approach. In the same way that a spreadsheet defines the rules for a complex set of repetitive calculations, a Style Sheet defines the rules for complex layout. Once these rules are defined, non-typesetters can quickly achieve typesetter-quality results simply by applying or tagging each paragraph as a Headline, Sub-Head, etc. Style sheets let even non-typesetters achieve typeset

Illustration of nozzle produced in AutoCAD. Line-Art can also be brought in from Monitor Graphics EE CAD. DXF compatible CAD packages, Lotus 123, GEM Draw or GEM Graph. Images can be brought in from PC Paintbrush, GEM Paint, MicroTek, Dest, and other scanners.

Figure 7

FPBM image of a different laser-printed page combining text with line drawings — file SCOPFPBM.

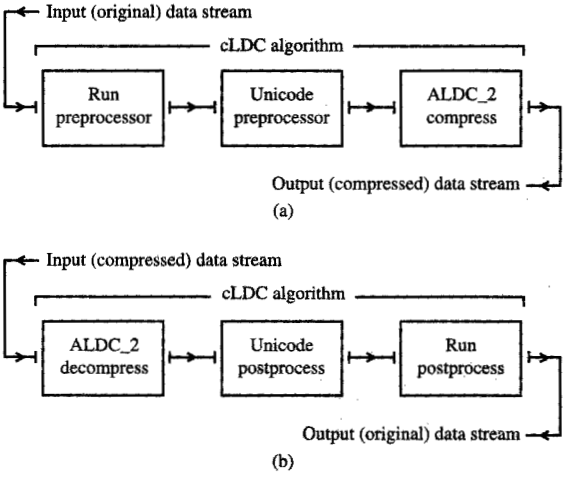


Figure 8

Compression/decompression block diagrams—cLDC_2 algorithm (ALDC_2 denotes 1024-byte history): (a) Compression; (b) decompression.

Table 3 shows results for both ALDC_1 and BLDC_1 algorithms (512-byte history size), on both 600-dpi and 1200-dpi-resolution versions of all the bitmap test image data files. Original sizes for these were about 4 MB at 600-dpi resolution and about 16 MB for the 1200-dpi versions. BLDC_1 shows an improvement over ALDC_1, ranging from about 1.32 to 2.76 times better on 600-dpi data. On 1200-dpi data, improvement ranges from about 1.32 to 3.34 times better.

The theoretical maximum compressions attainable from ALDC and BLDC are about 90:1 and 2700:1 respectively, but for most laser printers used in commercial business applications like these, typical data compression ratios achieved with BLDC_1 are likely to range from 7:1 to 20:1 at 600 dpi, and from about 10:1 to 40:1 at 1200 dpi. The improved compression at the higher resolutions is largely due to the preprocessor, which is itself more effective on longer run lengths.

cLDC general-purpose extension of ALDC

BLDC demonstrated the effectiveness of the preprocessor concept, but the scope of the algorithm was restricted to data for which the preprocessor was designed. The ALDC algorithm is still available from a BLDC chip implementation, by disabling the preprocessor. However, this requires some knowledge of the type of data being processed, and this may not always be available.

Recent work has thus been directed to automatic pre/postprocessors, which monitor a data stream and recode data only if it is advantageous to do so. A preprocessor monitors the input data, and its corresponding postprocessor monitors the output. Each uses the same heuristic to determine whether or not it operates. When inactive, they simply pass the data unchanged, in transparent fashion. Multiple pre/postprocessors can then be cascaded, to extend the range of data types over which an improvement can be obtained without requiring any action or knowledge about the data type from the using system.

An initial implementation of such an algorithm, called cLDC, is shown in **Figure 8**. This algorithm employs two cascaded pre/postprocessors, one designed to recode a run of identical byte values, the other to detect and recode Unicode-like data sequences. Unicode [13] is used by the Java language and other Web-based applications. It is likely to become a universal text-coding standard as the

Table 3 Image bitmap data compression ratios (ALDC_1/BLDC_1, 600/1200 dpi). Because these files are all full-page laser printer bitmap data, compressed as a single object, the compression ratio is an average for the entire page image.

File name	*ALDC_1/BLDC_1 (at 600 dpi)*	*ALDC_1/BLDC_1 (at 1200 dpi)*
1040FPBM	(9.24:1)/(22.18:1)	(13.81:1)/(41.69:1)
ABILFPBM	(10.48:1)/(28.94:1)	(16.68:1)/(55.69:1)
BWP1FPBM	(4.56:1)/(6.55:1)	(6.23:1)/(8.86:1)
BWP2FPBM	(4.37:1)/(6.28:1)	(5.97:1)/(8.19:1)
BWP3FPBM	(4.73:1)/(6.22:1)	(6.68:1)/(8.82:1)
BWP4FPBM	(5.25:1)/(7.55:1)	(7.78:1)/(12.35:1)
FONTFPBM	(12.08:1)/(28.78:1)	(18.07:1)/(52.68:1)
GOLFFPBM	(12.18:1)/(20.11:1)	(21.68:1)/(37.71:1)
MEOWFPBM	(15.08:1)/(24.22:1)	(20.36:1)/(43.77:1)
MOREFPBM	(7.80:1)/(15.65:1)	(13.87:1)/(22.24:1)
SCOPFPBM	(5.59:1)/(9.33:1)	(8.04:1)/(17.15:1)
SPRLFPBM	(8.27:1)/(12.79:1)	(12.87:1)/(23.88:1)
STARFPBM	(5.35:1)/(8.12:1)	(8.46:1)/(16.47:1)

Table 4 ALDC_2 vs. cLDC_2 compression results on CALGARY vs. U_CALGARY files. Files PAPER1 through PAPER6 are concatenated to form one, and the entire corpus is concatenated in sequence to form the file ALL.

File name	*ASCII*	*ALDC_2*	*cLDC_2*	*Unicode*	*ALDC_2*	*cLDC_2*
BIB	111,261	59,524	59,525	222,524	83,344	59,533
BOOK1	768,771	452,776	452,816	1,537,544	615,344	452,786
BOOK2	610,856	315,114	315,227	1,221,714	436,006	315,122
GEO	102,400	78,308	78,440	102,400	78,308	78,440
NEWS	377,109	218,653	217,513	754,220	310,198	218,662
OBJ1	21,504	11,265	11,248	21,504	11,265	11,248
OBJ2	246,814	106,574	106,534	246,814	106,574	106,534
P1-P6	245,231	128,481	128,530	490,474	177,940	128,535
PIC	513,216	63,531	59,518	513,216	63,531	59,518
PROGC	39,611	18,816	18,929	79,224	26,304	18,827
PROGL	71,646	26,136	26,045	143,294	37,868	26,146
PROGP	49,379	17,942	17,701	98,760	26,417	17,950
TRANS	93,695	41,963	41,798	187,392	60,555	45,131
ALL	3,251,493	1,538,420	1,533,133	5,619,080	2,033,071	1,537,769

internationalization and standardization of software and Web-based applications and e-business proliferate.

The run preprocessor simply retains the three predecessor data byte values from its input stream. If these are ever identical, it recodes by discarding subsequent data bytes while the identicality persists.

An eight-bit counter is incremented for each byte discarded, and it is inserted into the recoded data output when the run terminates. As for BLDC, if the run is long, a byte with the maximum count value is output, and a fresh count begins. Long runs are thus dramatically compressed, but they are also never seen by the ALDC compressor. The history remains filled with more diverse data, and the string matches upon which LZ1 compression depends are likely to occur more frequently once the run ends.

The Unicode standard uses two bytes to encode each character, and for many text data files this results in a data stream in which every other byte value is identical. ALDC does compress Unicode versions of such text files better than ASCII versions, but because the source file is twice as large to begin with, the end result is still about 40% larger than for ASCII. The cLDC Unicode preprocessor maintains the preceding nine bytes of data from its input data stream, and operates if the five even-ordered predecessor byte values are identical. Recoding is then accomplished by taking input data two bytes at a time instead of one, discarding the even-ordered byte while this identicality persists. A reserved byte value is inserted into the recoded output to signal the end of such a Unicode-like data sequence. Originally, identicality of three even-ordered predecessors was used as the

Table 5 Bitmap image compression ratios (ALDC_2/BLDC_1/cLDC_2 at 600/1200 dpi). Because these files are all full-page laser printer bitmap data, compressed as a single object, the compression ratio is an average for the entire page image.

File name	*Image resolution 600 dpi (ALDC_2)/(BLDC_1)/(cLDC_2)*	*Image resolution 1200 dpi (ALDC_2)/(BLDC_1)/(cLDC_2)*
1040FPBM	(17.12:1)/(22.18:1)/(19.55:1)	(14.57:1)/(41.69:1)/(33.14:1)
ABILFPBM	(20.46:1)/(28.94:1)/(24.89:1)	(17.24:1)/(55.69:1)/(44.46:1)
BWP1FPBM	(5.90:1)/(6.55:1)/(6.08:1)	(6.41:1)/(8.86:1)/(8.28:1)
BWP2FPBM	(5.53:1)/(6.28:1)/(5.75:1)	(6.36:1)/(8.19:1)/(7.62:1)
BWP3FPBM	(6.06:1)/(6.22:1)/(6.25:1)	(6.99:1)/(8.82:1)/(8.72:1)
BWP4FPBM	(7.62:1)/(7.55:1)/(7.85:1)	(8.04:1)/(12.35:1)/(11.83:1)
FONTFPBM	(20.77:1)/(28.78:1)/(24.05:1)	(19.01:1)/(52.68:1)/(43.72:1)
GOLFFPBM	(18.05:1)/(20.11:1)/(20.75:1)	(22.46:1)/(37.71:1)/(39.07:1)
MEOWFPBM	(17.92:1)/(24.22:1)/(23.75:1)	(27.56:1)/(43.77:1)/(36.22:1)
MOREFPBM	(14.21:1)/(15.65:1)/(15.44:1)	(14.17:1)/(22.24:1)/(25.80:1)
SCOPFPBM	(7.75:1)/(9.33:1)/(8.19:1)	(8.69:1)/(17.15:1)/(13.71:1)
SPRLFPBM	(9.63:1)/(12.79:1)/(10.92:1)	(13.25:1)/(23.88:1)/(18.60:1)
STARFPBM	(7.92:1)/(8.12:1)/(8.20:1)	(8.73:1)/(16.47:1)/(16.78:1)

condition, but English ASCII text files, with words such as "inimitable," then invoked preprocessor operation, so it was extended to five to reduce occurrences of this condition.

The hardware to implement both pre/postprocessors is trivial, requiring less than 10% of the CMOS chip area of the ALDC CRAM engine itself. Speeds of such processors are faster than ALDC in the same technology.

Table 4 shows a comparison between ALDC_2 and cLDC_2 (using 1024-byte history) on each of the individual files in the CALGARY suite and the U_CALGARY suite. The latter is the same as CALGARY except that ASCII text files are replaced with their Unicode versions.

Table 5 shows a comparison of cLDC_2 with BLDC_1 and ALDC_2 for the suite of FPBM binary image files at 1200 dpi. At the time BLDC was designed, the preferred history size was 512 bytes, but subsequent density improvements in the CMOS process have made it possible to use 1024 as the preferred history size, so cLDC_2 and ALDC_2 are used.

Conclusions

ALDC hardware implementations are easily the fastest available and are also easily the least expensive in terms of silicon area. The current CRAM designs achieve 100-MB/s sustained data compression or decompression speeds (measured as input to a compressor or output of a decompressor), for a silicon area in IBM's production CMOS 5 process of less than 10% of a small chip (7 mm × 7 mm). The newer CMOS 6 and CMOS 7 processes will reduce the area even further and provide additional increases in speed.

The compression approaches the effectiveness of more complex, two-pass software algorithms, yet when compared to software implementations using similar process technology, it is more than 1000 times more cost-effective, on a silicon area/speed product basis. The small size also allows it to be readily integrated on-chip with other functions (a microcontroller for example), and as such it is well suited for incorporation into many different kinds of computer storage peripheral devices.

The BLDC extension requires little additional silicon area, extending the applicability of these designs to binary bitmap image data compression, in addition to the general-purpose lossless LZ1 capability.

The cLDC extension also takes little additional silicon area and is in many ways superior to BLDC, since it automatically extends the capability of the general-purpose ALDC algorithm into the bitmapped image domain by performing almost as well as BLDC, and also considerably improves performance on Unicode data streams.

The CRAM compression technology is clearly able to cover an extremely wide range of applicability. Its small size allows integration into the smallest portable, hand-held, or wireless applications, where it is much faster and consumes far less power than software. It is viable for integration even within "smart" credit-card devices themselves. At the other end of the spectrum, this same design and algorithm are already capable of providing more than enough throughput for our most powerful mainframes and network file servers. Using the more advanced IBM CMOS 6 and CMOS 7 technologies, we can easily fit multiple engines onto a single chip. In turn, this allows us to design ALDC compression systems which have sustained throughput in the gigabyte-per-second range, if required, and should effectively meet system storage and networking application requirements into the next millennium.

**Trademark or registered trademark of Sun Microsystems, Inc. or The Open Group or X/Open Company Ltd.

References

1. A. Lempel and J. Ziv, "On the Complexity of Finite Sequences," *IEEE Trans. Info. Theory* **IT-22,** No. 1, 75–81 (1976).
2. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **IT-23,** No. 3, 337–343 (1977).
3. J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Trans. Info. Theory* **IT-24,** No. 5, 530–536 (1978).
4. D. J. Craft, "Method and Apparatus for Compressing Data," U.S. Patent 5,652,878, July 7, 1997.
5. J. Cheng, L. M. Duyanovich, and D. J. Craft, "A Fast, Highly Reliable Data Compression Chip and Algorithm for Storage Systems," *IBM J. Res. Develop.* **40,** No. 6, 603–613 (1996).
6. R. P. Brent, "A Linear Algorithm for Data Compression," *Austr. Computer J.* **19,** No. 2, 64–68 (1987).
7. J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution," *J. ACM* **29,** No. 4, 928–951 (1982).
8. T. C. Bell, "Better OPM/L Text Compression," *IEEE Trans. Commun.* **COM-34,** No. 12, 1176–1182 (1986).
9. J. Cheng, D. J. Craft, L. J. Garibay, and E. D. Karnin, "Efficient Ziv–Lempel LZ1 Data Compression System Using Variable Code Fields," U.S. Patent 5,608,396, March 4, 1997.
10. Standard QIC-154, "Adaptive Lossless Data Compression (ALDC)," Revision A, at ⟨*http://www.qic.org*⟩, March 10, 1994. (See also standards ECMA-222, ISO/IEC 15200, and ANSI x3.280-1996.)
11. Calgary Corpus, available from the University of Calgary, Canada, via anonymous file-transfer protocol from ⟨*ftp.cpsc.ucalgary.ca*⟩.
12. D. J. Craft, "Dual Stage Compression of Bit Mapped Image Data Using Refined Run Length and LZ Compression," U.S. Patent 5,627,534, May 6, 1997.
13. On-line information about the Unicode standard available from the Unicode Consortium at ⟨*http://www.unicode.org*⟩.

Received February 3, 1998; accepted for publication May 4, 1998

David J. Craft *IBM Microelectronics Division, 11400 Burnet Road, Austin, Texas 78758 (dunstan@us.ibm.com).*

Mr. Craft received a B.Sc. in physics from Imperial College, London University (U.K.), in 1963, joining IBM in 1965. He worked on a number of advanced development projects at the IBM laboratory in Hursley (U.K.) until 1978, then in laboratories in Boulder, Tucson, and Austin (U.S.). Mr. Craft holds 23 issued patents; he has seven recent applications in process. One 1974 patent (U.S. 3,818,447), on a serial bus arbitration mechanism, is now the basis for both the PC "plug and play" hardware standard and the widely used J1850 and CAN serial interface protocols for automotive, industrial, and consumer electronics applications. A recent patent (U.S. 5,652,878) on a fast CAM (content-addressable memory)-based LZ1 compressor design is the basis for IBM's ALDC range of chip products. Mr. Craft has received two IBM Special Contribution Awards and an IBM Outstanding Technical Achievement Award, the latter for his work on fast data compression hardware designs.

[[Page 746 is blank]]