Design considerations for the ALDC cores

by M. J. Slattery
F. A. Kampf

The IBM adaptive lossless data compression (ALDC) family of products uses a derivative of Lempel–Ziv encoding to compress data. Several variables affect the compression performance of the ALDC algorithm: data content, history size, and data extent. As ALDC compression is integrated into different applications, restrictions are placed upon these variables that affect the overall compression performance of the system. More complex applications require further support for higher-order data structures such as variable-length segments, error recovery, and expansion. The IBM Blue Logic ALDC and embedded lossless data compression (ELDC) cores have been developed to work in these application environments. These cores and the issues associated with integrating data compression into a system are discussed.

Introduction

As a general compression technique, the Lempel–Ziv algorithm [1] integrates well into systems required to handle many different data types. This algorithm processes a sequence of bytes by keeping a recent history of the bytes processed and pointing to matching sequences within the history. Compression is performed by replacing matched byte sequences with a copy pointer and length code that together are smaller in size than the replaced byte sequence. Once the history is filled, it adapts to incoming data to represent the last N bytes. Matching byte sequences that propagate beyond the extent of the history continue to match until the maximum codable length has been reached. The amount of compression that occurs is determined by the data content, the history size, and the amount of data processed.

When data compression is integrated into a system, that system applies structure to the data. Depending upon the system, the complexity of the structure varies. For instance, data stored on hard disks and tapes is file-oriented, but for hard disks, each file is a collection of discrete clusters. The entire file is accessed on the storage media independent of the underlying structure. In communication systems, data is partitioned into small blocks, packets, or frames prior to being transmitted. Along the way, the data blocks may be multiplexed with other data blocks before reaching their final destination. In each of these systems, the structure is created to help manage the data.

Invariably, this results in a tradeoff of less favorable compression performance for improved data access. In situations where the data is partitioned into segments, effective compression decreases when the segments are compressed individually. If sequential access to the data segments can be guaranteed, segmented compression can allow for the retention of the compression context between compression operations. This enhances the compression ratio by taking into account the inherent dependency between consecutive segments of data. Other

0018-8646/98/$5.00 © 1998 IBM

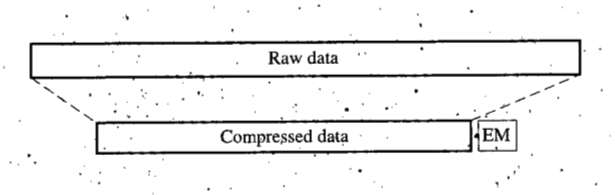

Figure 1

Compression.

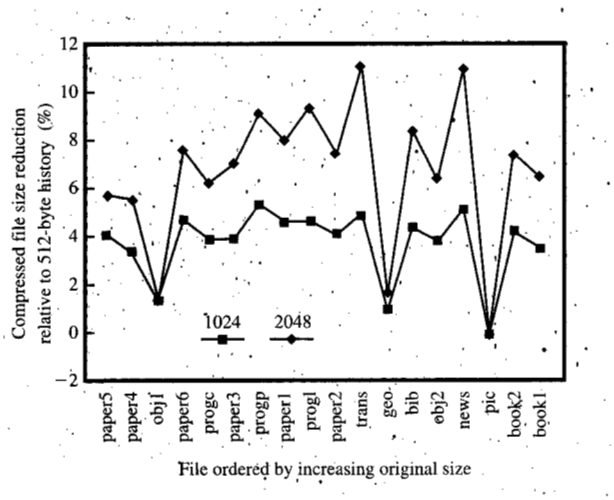

Figure 2

Effect of increasing history size.

system considerations, such as variable-length segments, error recovery, and expansion, further complicate the integration of data compression into a system.

IBM has developed a family of products that employ a proprietary version of Lempel–Ziv encoding called IBM LZ1. The adaptive lossless data compression (ALDC) products use this proprietary hardware-encoding scheme to produce superior compression and data throughput. The ALDC and embedded lossless data compression (ELDC) cores in the IBM Blue Logic core library apply the ALDC algorithm to systems that use data structures to manage data. The ALDC and ELDC cores provide the flexibility required to handle the many different data compression situations presented in today's computer systems.

This paper describes the important factors associated with compressing data with the ALDC algorithm. It examines complications introduced when data is structured and discusses enhancements to compression by sharing compression contexts between segments. Finally, the new features of the ELDC core that address variable-length segments, error recovery, and minimization of expansion are presented.

Factors affecting performance

The first ALDC products (ALDC-5S, -20S, and -40S) [2] compress data as one continuous stream of data. This data is compressed by analyzing successive bytes until all bytes have been processed. As each byte is processed, it is compared against the most recent bytes in the history. As bytes are compared, the ALDC engine keeps track of consecutive matching bytes. When the longest matching sequence of bytes is determined, a code word is inserted into the compressed data stream that points to the location and the length of the matching sequence. If no matches are found, the data is coded as a literal and also inserted into the compressed data stream. Compression is realized when byte sequences are replaced by smaller code words. The details of the ALDC algorithm are presented in a companion paper [3] by Craft.

When data is compressed as a continuous stream, no indication of the former data structure is maintained in the compressed form. That is to say, any boundary in the original data will be indistinguishable in the compressed data. To retrieve any bytes within the original data, it will be necessary to decompress the entire preceding data structure.

Figure 1 shows a sample compression. The end marker (EM) denotes the end of the compressed data stream.

The achievable compression performance depends most significantly upon the content of the data being compressed. If many long, matching byte sequences are encountered, compression performance will be maximized. However, if entirely random data dominates, creating only literal code words, the compressed data stream will expand 12.5%. Unfortunately, in most cases, the application cannot control the nature of the data it is working with.

One factor the application can control is the size of the history. The size of the history affects compression: The larger the history, the more sequences that are available to be referenced. Although the average code-word size does increase slightly as the history depth increases, effective compression can increase. In the ALDC cores, the history buffer can be configured in sizes of 512, 1024, and 2048 bytes. **Figure 2** shows the impact of the history size on compression for a group of files known as the Calgary Corpus [4], arranged by increasing original file size. The Calgary Corpus represents a range of typical file types that would appear in a computer system. In most cases, the larger history enables a higher compression ratio. Although this is dependent upon the type of data

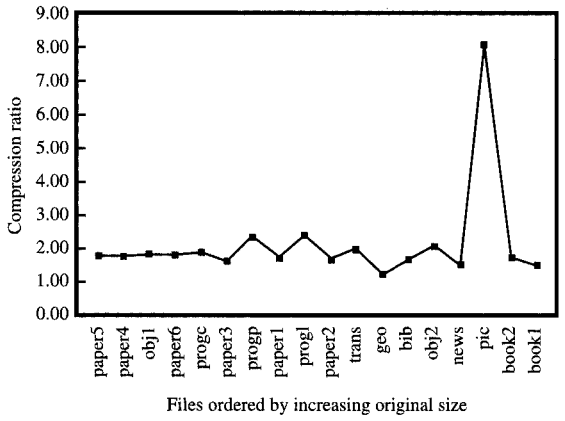

Figure 3

Compression results for the Calgary Corpus.

used, diabolical data types are minimally affected by a larger history buffer size.

The final factor that affects compression performance is the size of the data to be compressed. The graph in **Figure 3** shows a plot of compression ratios for the Calgary Corpus collection of files. The content of the data modulates compression performance much more than the file size. For large volumes of data, the ALDC algorithm is relatively independent of the number of bytes processed. Size becomes significant when the byte counts approach the depth of the history. This effect is discussed further in the next section.

Segmented compression

In complex systems requiring higher levels of organization, data is separated into smaller, more manageable segments. Segmented data can be found everywhere, from communication systems in CSU/DSUs to networking protocols (Ethernet, Token Ring, and ATM) and even personal computer file systems (hard disks, CD-ROMs, and tape drives). Partitioning the data permits structure to be preserved during compression and compressed data to be multiplexed after compression. Segmented compression divides the raw data prior to compression. Each system presents a different set of requirements that affect data organization and data compression.

Figure 4 demonstrates segmented compression. The raw data is partitioned into either fixed-length segments or variable-length segments. The compressed segment sizes vary depending upon the specific data contained within each segment.

One factor that inhibits effective data compression of segmented data is the history. When each segment begins

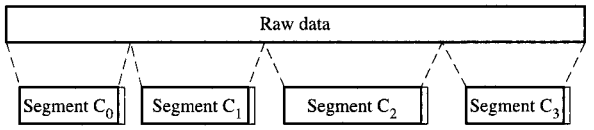

Figure 4

Segmented compression.

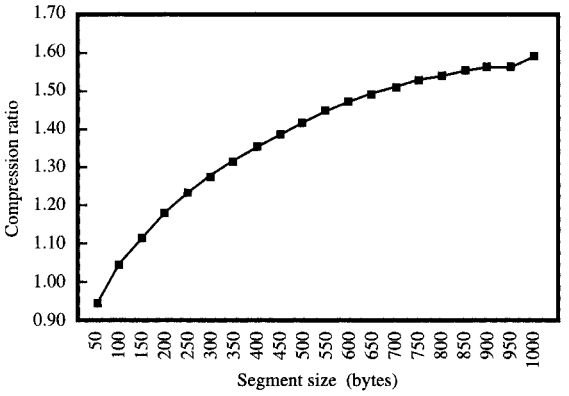

Figure 5

Calgary Corpus file *paper1* compressed with various segment sizes.

with an empty history, fewer byte sequences exist to compare against during the beginning of every segment. As a result, the early compressed data stream consists of more literals than copy code words. Typically, compression is minimal while the history is being filled.

In conjunction with the history, the segment size also affects compression. As the segment size approaches the depth of the history, the compression performance typically degrades. The adaptive nature of the ALDC cores affects their ability to compress segmented data because the probability that the history will contain any long, matching byte sequences is greatly reduced.

In **Figure 5**, the file *paper1* from the Calgary Corpus has been compressed with different segment sizes. Notice how the compression ratios at the various segment sizes are all less than 1.60:1, whereas *paper1* compressed as a continuous stream of data achieves a compression ratio of 1.79:1. As the segment size approaches the size of the original data, the effect of partitioning the data is minimized. This reduction in compression ratio is a result of starting each segment with an empty history, which

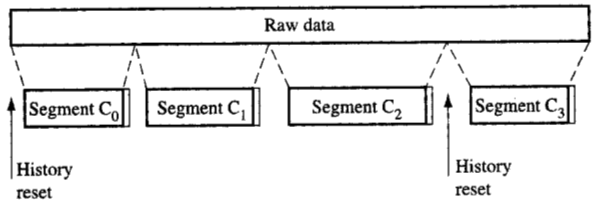

Figure 6

Dependency between compressed data segments.

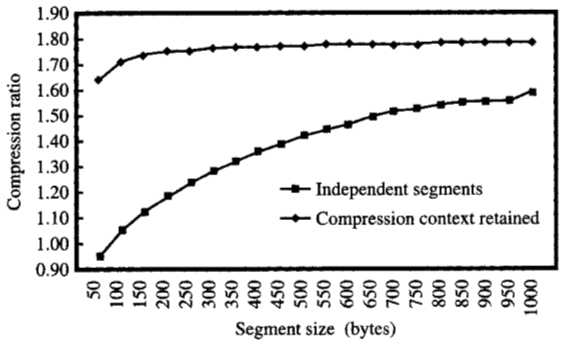

Figure 7

Calgary Corpus file *paper1* compressed with various segment sizes with the compression context retained between segments.

creates fewer matching byte sequences and truncating matching sequences at the segment boundary. By the time the compression ratio begins to approach its aggregate value, the segment is completed. There is not enough data to reach peak efficiency with small segments.

Intersegment dependencies

As seen in the discussion of segmented compression, the impact of partitioning data is significant. Inherent in that discussion is the presumption that all segments can be decompressed independently of the others. In some systems, segment ordering is required. A given segment may have no meaning until the previous segments are decoded. If this is the case, the ALDC cores can take advantage of this relationship between segments to improve effective compression.

For instance, when accessing a file on a tape drive, it is not efficient to decompress the entire file system simply to retrieve one file. However, for a network system in which a relationship between the packets already exists in the packet ordering, that relationship can be used to improve compression performance. If the compression context between segments is saved, successive segments can refer to byte sequences in the previous segment and realize compression immediately. Since the first packet must be decoded first, the second packet can use the history from the end of first packet. It will already be available when the second packet is decoded. The compression context includes the contents of the history and all associated pointers and match counters.

Segmented compression can allow part of the compression context to be retained from one compression operation to another. Since it is assumed that a relationship exists among segments, retaining access to the history buffer between operations significantly improves the compression ratio of each segment.

For example, consider **Figure 6**, which shows original data partitioned into four segments. Assume that the history buffer is reset between original segments C_2 and C_3. In order to decompress segment C_1, it is necessary to decompress both segments C_0 and C_1, in order. Likewise, it is necessary to decompress segments C_0, C_1, and C_2, in order, to access data within the third segment. The fourth segment is independent of all previous segments, so it may be decompressed by itself. As with compression without segments, it is necessary to decompress all of the previous data from the last history reset to retrieve a given byte within that segment.

In **Figure 7**, *paper1* is again compressed with various segment sizes. The new data represents *paper1* compressed while retaining the compression context between segments. The effect of the end-of-segment marker and broken-byte-sequence matches can be seen at very small segment sizes. The end-of-segment marker adds thirteen bits to the compressed data stream, causing minor expansion. Broken byte sequences result in multiple copy code words of smaller copy lengths or literals replacing copy code words of larger matches. As segment sizes increase, the compression ratio approaches the continuous-stream compression ratio much earlier than independent segments do. The result is almost constant compression independent of segment size. Allowing the compression context to be retained between segments improves ALDC's ability to compress segmented data.

Extending segmented compression

As applications using data compression evolve, more intricate data structures are required. In late 1997, the Linear Tape–Open (LTO) [5] alliance, consisting of the Hewlett-Packard, Seagate, and IBM companies, proposed a standard to unite a fragmented tape-drive industry. The Linear Tape–Open Data Compression (LTO–DC)[1]

[1] Linear Tape–Open Data Compression specifications (not yet released).

specification defines a format with variable-length segments and provides for methods to minimize expansion and recover from system errors. This advanced form of partitioning is supported by the new embedded lossless data compression (ELDC) core.

In this format, raw data can be partitioned into relevant segments, such as blocks, clusters, files, etc. Each raw segment can be compressed to form a record, which may be collected into a formatted block. Formatted block size is programmable and can be configured to comply with the LTO specifications. Additionally, the LTO specification defines the generation of decompression access points. Access points within each compressed formatted block correspond to a location at which the history was reset. Extraction of a record from the compressed data is accomplished by decompressing from the preceding access point. During compression, the ELDC core tracks record and formatted block boundaries and automatically resets the history to create an access point. The location of the access point within the compressed data stream is provided through status registers.

The ELDC core also reduces the overhead for error recovery by dividing records into one or more subsegments, known as *bursts*, as shown in **Figure 8**. A compressed burst is the smallest identifiable block of compressed data generated by the ELDC core, and is padded to a four-byte boundary. Each burst is terminated by an end-of-burst (EOB) control code. The final burst is terminated by either an EOB or the end-of-record (EOR) control code. This feature provides the ability to correlate compressed data blocks to data bus transfers, independently of the overall record size. If a very large record is divided into smaller bursts, a bus parity error within a single bus transfer does not necessitate the recovery of the entire record. Retransmission and compression can occur on the failed burst boundary.

The artifact of data expansion is also addressed by the ELDC core. The LTO–DC format defines a method of indicating two alternate compressed data modes within the compressed data stream. The first mode is composed of ALDC compressed data, and the second mode is composed of raw data. The ELDC core switches optimally between the two modes to minimize any potential data expansion.

Experimentation with raw data of varying randomness graphed in **Figure 9** shows the effects of the mode-changing abilities of the ELDC core. The compression ratio obtained from the original ALDC compression algorithm decreases to 0.89:1 (12.5% data expansion) as the randomness of the raw data approaches 100%. However, when the ELDC scheme-swapping algorithm is applied to the same data, the compression ratio stabilizes around 1:1 as the randomness of the raw data increases.

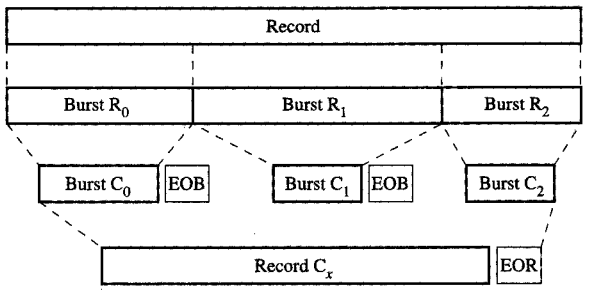

Figure 8

Compression of a record.

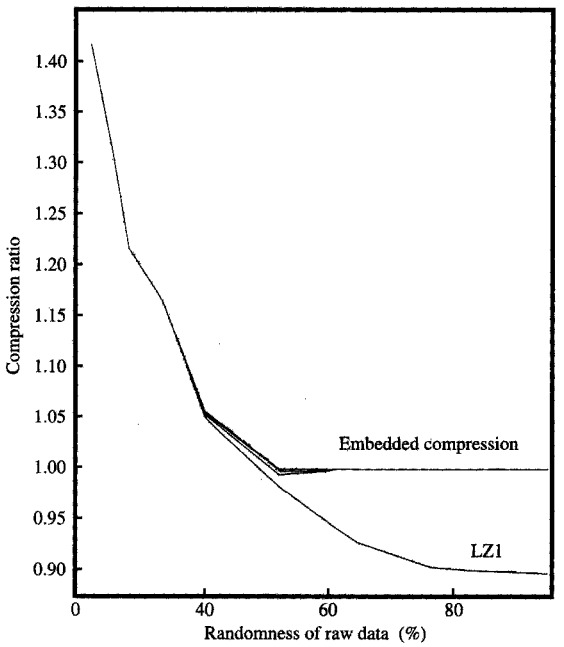

Figure 9

LZ1 vs. embedded compression performance with swapping.

Summary

With the increasing need to integrate as much function as possible into VLSI applications, the ALDC and ELDC cores provide flexible, lossless data compression solutions. The system integrator must consider the size of the history and how the data is managed to optimize compression performance in a particular application. By providing automatic data segmentation and history control, the

ALDC core provides a compression/decompression solution for many system environments. The ELDC core extends this architecture to work with variable-length segments with burst controls, error recovery, and expansion minimization to address the requirements of advanced systems. The ALDC and ELDC cores continue to evolve to meet the needs of today's complex systems.

Acknowledgments

This paper would not have been possible without the guidance and inspiration of Joan Mitchell, the enthusiastic interest of Stuart Burroughs, the support of Bill Lawrence, the excellent technical work of David Craft, Julie Cubino, Rudy Farmer, Rob Gibson, Joe Hamel, and Oscar Strohacker, and the unqualified support of the authors' managers, Mark Merrill and Ted Lattrell.

References

1. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **IT-23**, 337–343 (1977).
2. *http://www.chips.ibm.com/products/aldc/index.html.*
3. D. J. Craft, "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM J. Res. Develop.* **42**, 733–745 (1998, this issue).
4. T. C. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression," *Computing Surv.* **21**, 557–591 (December 1989).
5. *http://www.lto-technology.com.*

Received May 28, 1998; accepted for publication October 23, 1998

Michael J. Slattery *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (mslatter@us.ibm.com).* Mr. Slattery graduated from the University of Notre Dame with a B.E. degree in electrical and computer engineering. He joined the IBM Microelectronics Division in 1990 and has worked in VLSI failure analysis, polyimide films and via process engineering, encryption product design, adaptive and lossless data compression product design, arithmetic coding for VLSI implementations, and 2D graphical acceleration.

Francis A. Kampf *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (kampf@us.ibm.com).* Mr. Kampf attended Temple University, earning a B.S. in engineering, *magna cum laude*, in 1987. He joined IBM Kingston in 1988 and participated in the development of an FDDI-based interconnect controller. His continued development effort in the communications interconnect arena resulted in a cooperative effort with the IBM Zurich Research Laboratory and the demonstration of a prototype gigabit WAN/LAN at Telecom'91. He joined the newly formed IBM POWERparallel group in 1992 and participated in the development of the Scalable POWERparallel (SP) line of massively parallel computers. Mr. Kampf's work on the communication subsystem has led to eleven pending patent applications. In 1996, he joined the IBM Blue Logic core development area at IBM Burlington to develop data compression cores. His work includes the development of the JBIG–ABIC and ALDC/BLDC and ELDC cores.