Performance as a function of compression

by F. A. Kampf

This paper discusses the performance of bilevel-image arithmetic coders, ABIC and JBIG, and Lempel–Ziv string compressors, ALDC and BLDC. Images are analyzed for typical and worst-case throughput and latency as a function of compression. A relationship between the compressibility of an image and the throughput performance of the compression algorithm is demonstrated. Generally, throughput performance of the bilevel-image arithmetic coders decreased as image entropy increased. Inversely, the bilevel-image string compressor (BLDC) revealed that increased entropy improved throughput performance. Experimental results based on hardware implementations have been provided and analyzed.

Introduction

The performance of data compression algorithms is of concern to the system architects. This paper discusses the performance characteristics in relation to the compression achieved by the different lossless data compression algorithms available as IBM Blue Logic cores. Two performance characteristics of concern that have an effect in overall system performance are throughput and latency.

The Adaptive Bilevel Image Compression (ABIC) algorithm performs lossless data compression on bilevel images [1]. The algorithm utilizes the Q-coder adaptive binary arithmetic coder to produce a finite-precision binary fraction that uniquely identifies the sequence of bits in the raw-data image [2, 3]. The coder processes the individual bits of the raw data stream, calculating the probability of the symbol (binary bit value) on the basis of the context obtained from a template applied to the raw data. The probability represents the prediction of the more probable symbol (MPS) [or, inversely, the less probable symbol (LPS)] to be encountered next. Maintaining the interval calculation within the order of unity requires renormalization of the code string and interval size. The processing of an MPS potentially produces one renormalization, and the processing of an LPS can potentially produce up to 12 renormalizations. Each renormalization produces one bit of coded data.

The Joint Bi-level Image Experts Group (JBIG) defined a lossless data compression algorithm suited for bilevel images that utilizes the QM-coder adaptive binary arithmetic coder [4, 5]. The QM-coder produces an infinite-precision binary fraction uniquely identifying the sequence of raw data bits. The IBM Blue Logic core implementations of the JBIG and ABIC macros [6] utilize the Qx-coder, which incorporates both the Q-coder and QM-coder [7].

The adaptive lossless data compression (ALDC) algorithm [8, 9] is the IBM implementation of Lempel–Ziv compression algorithm 1 (LZ1) [10]. The LZ1 algorithm defines a fixed-size sliding window, conceptually a history of the previous symbols processed, used to perform pattern matching against the incoming data stream. The ALDC algorithm defines a symbol to be one byte of data, and supports history buffer sizes of 512, 1024, and 2048 bytes. Sequences of bytes that match sequences maintained within the history buffer are represented in the coded data as copy-pointer and match-length code words.

0018-8646/98/$5.00 © 1998 IBM

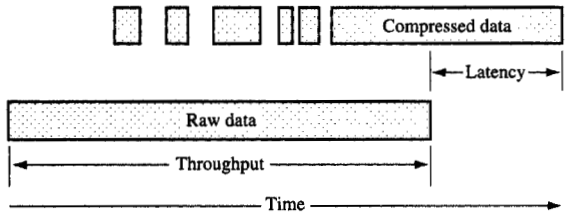

Figure 1

Compression latency.

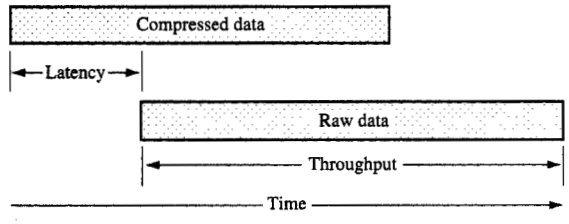

Figure 2

Decompression latency.

Bytes which cannot be included in matches are encoded as literals with a flag bit.

The ALDC compression algorithm has been extended with a bit-map-optimized preprocessor, referred to as BLDC, to enhance the algorithm's capabilities when compressing bilevel images [8, 9]. The BLDC preprocessor utilizes a run-length encoding scheme that defines byte-sized run-length codes which are readily processed by the ALDC core.

Any differences between these compression algorithms, such as wrappers and marker codes, have been disregarded in this study to permit a basic comparison of compression performance of the algorithms and implementations. Additionally, overhead due to system interfaces has been set aside because of the differences in implementations among the compression cores.

- *Throughput*

During data compression/decompression, the amount of data changes during the processing of the data stream according to the applied algorithm. To realistically compare different algorithms, the definition of throughput must be independent of the effects of the data compression process. Therefore, throughput as the rate at which raw data units (i.e., bits or bytes) are processed provides the relative performance figure from which comparisons can be made.

The rate of compressed data is related to the throughput by the compression ratio. In an ideal algorithm, the compression operation processes one data unit per algorithm cycle. Degradation from this ideal is due to characteristics of the algorithm and the implementation. To simplify this discussion, it is assumed that an implementation cycle (i.e., clock cycle) is equivalent to an algorithm cycle.

Data compression algorithms that utilize the arithmetic coder, such as JBIG and ABIC, are structured around a one-bit data unit. Therefore, the maximum performance level of these algorithms is one bit per cycle. Other algorithms which are based upon string matching (Lempel–Ziv), such as ALDC and BLDC, process a data unit of one byte, and the maximum performance level of these algorithms is one byte per cycle.

- *Latency*

Latency in the classic sense cannot be applied to data compression algorithms. The delay observed between the first unit of raw data and the first unit of compressed data cannot be considered as latency. If the raw data is highly compressible, the first unit of compressed data may be delayed a considerable amount of time. Also, owing to the nature of compression, the compressed data tends to be intermittent, with the possible result that the first unit of compressed data appears quickly, followed by a long pause. This typically has no real impact on system performance, since the raw-data rate is of overall importance. A pause in the processing of raw data is indicated by a lower throughput rate. On the basis of these factors, another view of latency has been considered.

Since performance is associated with the processing of raw data, and throughput accounts for the rate of the raw data, latency can be considered the delay before or after an operation when raw data is not being processed. The delay between the last unit of raw data and the last unit of compressed data is a period of time in which data compression is occurring, but the raw data has completed processing. This can be considered an impact to system performance. During this period, use of the compression core for the next operation cannot proceed. Therefore, as shown in **Figure 1**, compression latency is the time required to flush out the remaining buffered compressed data after processing the last unit of raw data.

During decompression operations, the last unit of raw data may not be produced from the compressed data stream until well after the last data unit has begun processing. This is indicative of highly compressed data sets. However, the delay between the first unit of raw

data following the first unit of compressed data can be considered latency, as shown in **Figure 2**. The greater the latency, the greater the delay before the next operation can commence.

• *Random images*

To make a fair comparison of the performances of different data compression algorithms, a set of patterns are needed that are not biased toward one algorithm. In this study, the set of patterns (i.e., images) were generated randomly and were of varying probability. This approach provides images of varying compressibility and does not create images that may be suited to a particular template organization or history buffer size.

The process of generating a random image utilized the given probability for each image to determine the value of each individual bit. As the probability of a black pel (bit equal to binary 1) approaches 0.5, the image becomes more random and less predictable, thus less compressible. And a probability of a black pel close to 0.0 provides a very uniform image that is highly predictable and compressible. **Figure 3** displays the effects of varying the probability of a black pel on the randomly generated images.

Arithmetic coders

Typically arithmetic coding algorithms process one bit of raw data during each cycle. Degradation from this ideal throughput performance occurs during the renormalization process. A bit of data is produced for each renormalization of the interval (A) and code (C) registers within the coder. However, under some conditions, multiple-bit renormalizations are required. Processing does not continue until all additional renormalizations have completed. Therefore, each extra (>1) renormalization incurred during processing can add additional cycles to the processing of the raw data, reducing the data rate. Experimentation has shown a relationship between the amount of renormalization and the compression ratio from which a correspondence can be drawn.

• *ABIC throughput*

The IBM Blue Logic core implementation of the ABIC algorithm processes and produces up to one bit per cycle. When extra renormalizations are encountered, the processing of input data bits pauses while the additional output bits are generated. The greater the number of multiple-bit renormalizations, the lower the data rate.

Three sets of random images were used to characterize ABIC throughput performance. Additionally, a set of test images were analyzed to determine the difference between actual images and random images. The normalized results

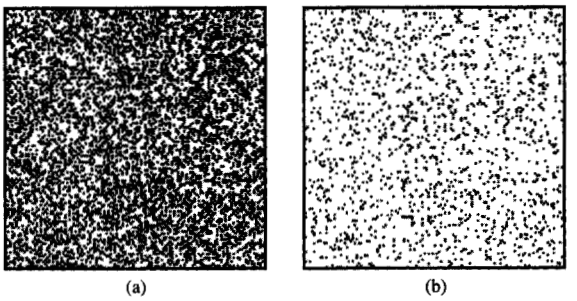

Figure 3

Examples of random images; probability = (a) 0.4; (b) 0.1.

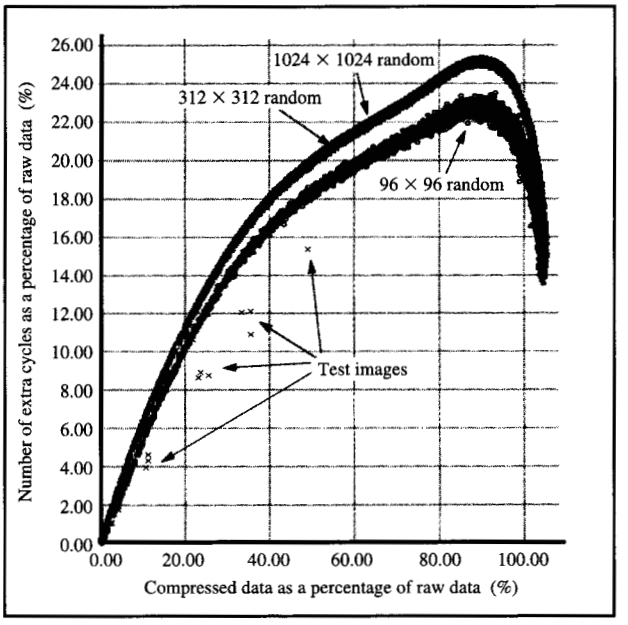

Figure 4

ABIC performance graph.

are graphically presented as a percentage of the raw data amount in **Figure 4**. Each point on the graph corresponds to the result of compressing these images and designates the number of extra renormalizations incurred during processing in relation to the resulting amount of compressed data.

According to the experimental results, the smallest random-image size appears to have incurred a lesser penalty from extra renormalization cycles. As random-

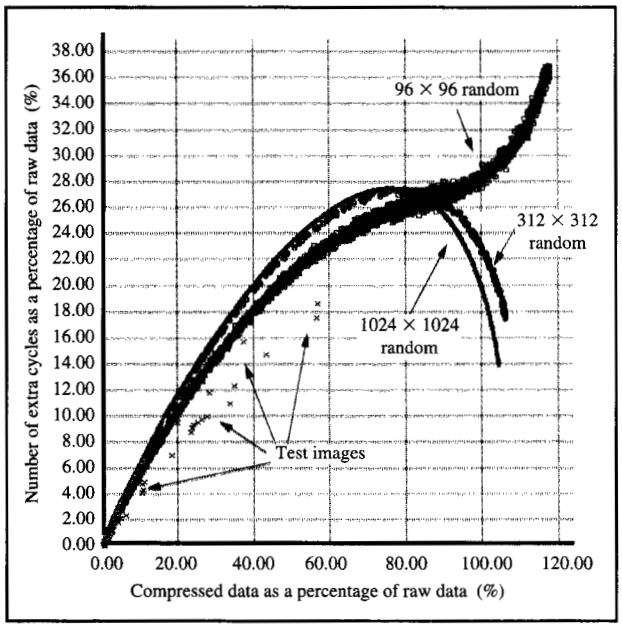

Figure 5

JBIG performance graph.

image size increases, the points solidify into a well-defined line. The worst-case performance encountered correlates to random images obtaining about 10% compression, where the data rate is at 80% of maximum (0.80 bit per cycle). All of the test images performed better than the random images in both data rate and compression ratio, as shown in the graph. Since the random images can be interpreted as noisy patterns, it is reasonable to predict that realistic images will generally perform better.

Given an image of 100% LPS, expansion is due only to the final flushing of the coding registers and generates no multiple-bit renormalization cycles. However, the worst expansion encountered during testing was found to be about 5% (compression ratio of 0.95:1). This occurred in all three random-image sizes, though the impact on performance decreased as the expansion increased. This behavior correlates to the coding inefficiency displayed by the Q-coder compression adaptation process [11].

Another expansion characteristic of the ABIC algorithm relates to the handling of a carry. The ABIC compression algorithm provides for the resolution of a carry within the decoder. A carry is placed within the compressed data stream by the encoder through a method known as *bit stuffing*. Bit stuffing occurs whenever a 0xFF byte is generated during compression, inserting a stuff bit before the next bit of coded data. This action inherently increases the amount of compressed data. However, only a sequence of MPSs will generate multiple 0xFF bytes, which in turn are highly compressible, obviating the impact of the stuff bit. Additionally, MPS renormalizations cannot produce multiple-bit renormalizations. Consequently, during high levels of compression, the increased frequency of MPSs reduces the number of multiple-bit renormalizations, lessening the impact on data rate. This is evident in the plotted experimental data showing all curves approaching the origin.

- *JBIG throughput*

The JBIG algorithm provides options for different templates, with a movable adaptive-template bit and indication of identical lines (typical prediction). The variations of template formats are obviated by the use of random patterns, causing all configurations to encounter the same probability. The use of typical prediction was not considered, since this requires comparison of each line and would cause significant overhead, resulting in reduced throughput. Essentially, to perform typical prediction, each line must be examined completely before processing can proceed, and the arithmetic coder state must be altered. Alternately, the state of the coder must be preserved, and it must be restored when a line is found to be typical. For the purpose of this study, the overhead of the JBIG header and floating marker codes was not counted. Only the amount of compressed data contained in the single stripe generated by the compression process was counted to determine the compression ratio.

As with ABIC, the IBM Blue Logic core implementation of the JBIG algorithm processes and produces up to one bit per cycle. When extra renormalizations are encountered, the processing of input data bits pauses while the additional output bits are generated. The greater the number of multiple-bit renormalizations, the lower the data rate.

The same three sets of random images and set of test images were used to characterize JBIG throughput performance. The normalized results are graphically presented as a percentage of the raw-data amount in **Figure 5**. The results, though generated with the JBIG three-line template, are indicative of both template configurations. Each point on the graph corresponds to the result of compressing these images and designates the number of extra renormalizations incurred during processing in relation to the resulting amount of compressed data.

As random-image size increases, the points solidify into a well-defined line. Compression ratios greater than 1.25:1 approach this line from below. However, for compression ratios less than 1.25:1, the characteristic is reversed. The worst-case performance encountered appears with the smallest random-image size, which produced an expansion of about 18% along with a data rate of 73% of maximum

(0.73 bits per cycle). This is in part due to the initial learning states of the QM-coder probability-estimation state machine. When an image is generally not compressible, the initial probability estimation of the QM-coder overcompensates away from the 50/50-probability state, which is subsequently encountered by continued processing. Small images may complete processing during a period when adaptation has not settled toward the 50/50-probability nontransient state indicative of uncompressible data.

Ignoring the upturned curve of the smallest random-image size, the worst-case performance encountered correlates to random images obtaining about 20% compression, where the data rate is at 78% of maximum (0.78 bits per cycle). All of the test images performed better than the random images in both data rate and compression ratio, as shown in the graph. Since the random images can be interpreted as noisy patterns, it is reasonable to predict that realistic images will generally perform better.

The worst expansion encountered during testing was found to be about 18% (compression ratio of 0.85:1). This occurred in the smallest of the random-image sizes, which also contained the highest penalty for renormalization. However, the JBIG algorithm reserves marker codes that can be imbedded into the data stream. To prevent confusion between marker codes and coded data, any occurrence of the escape byte (0xFF) in the coded data is expanded by appending a byte of 0x00 data. This convention can produce large expansion in some images.

Additional images were generated to evaluate the impact of byte stuffing. These images produce coded data containing all 1s when compressed, forcing the JBIG compression algorithm to add the stuff byte. The production of the stuff byte (eight additional output bits) creates a pause in processing and thus affects the data rate. The experimental results are displayed as "100% byte-stuffed images" in **Figure 6**. The curves created from the random and test images are included in the graph for reference. These results indicate how a small set of the pattern space can degrade JBIG compression significantly.

• *ABIC latency*

At the termination of the ABIC compression process, any remaining compressed data must be flushed. This process includes flushing any significant data within the code (C) register, the spacer bits, and up to seven coded bits awaiting output. Another consideration is that the last raw bit processed may cause a multiple-bit renormalization. Given that an LPS can incur at most 12 renormalizations, plus the 23 bits that can be flushed from the C register, the worst-case latency is no more than 35 bits. In the process of aligning data with byte boundaries, this may produce up to six bytes of data. Therefore, the worst-case latency of the compression algorithm is then 48 cycles.

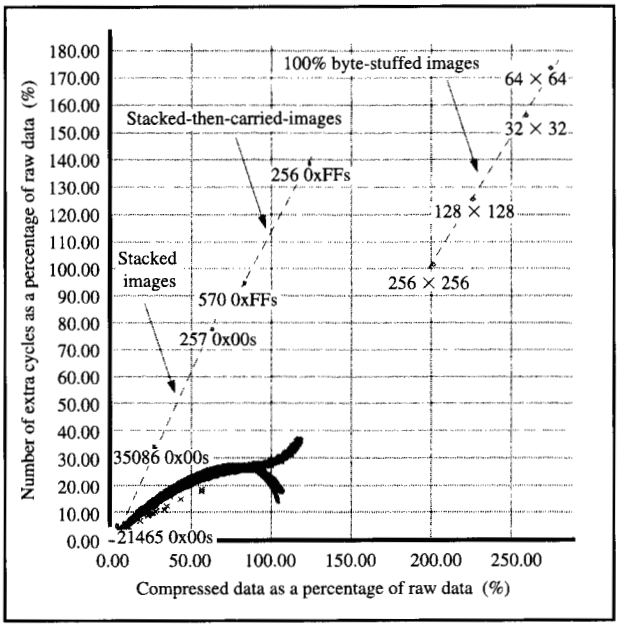

Figure 6

JBIG worst-case performance graph.

• *JBIG latency*

The JBIG compression algorithm also must flush any remaining compressed data from the registers at the end of the raw data stream. However, the JBIG algorithm produces an infinite-precision number which requires that the carry be resolved within the encoder. Owing to this requirement, coded data that can propagate a carry must be buffered until a carry is produced or propagation becomes impossible. The IBM Blue Logic core implementation maintains a stack count of 0xFFs when coded data is generated. Upon a carry, or the generation of a non-0xFF byte, the stack is emptied. When the stack is being emptied, processing cannot continue, reducing the data rate. If the output of the stack is a series of 0xFF (carry-propagated), byte stuffing will occur, adding additional cycles.

A set of diabolical images were generated to analyze the performance of processing patterns that compress with high stack counts. This set of images contain examples of compression both with and without a carry. To demonstrate the potential latency, any carries are

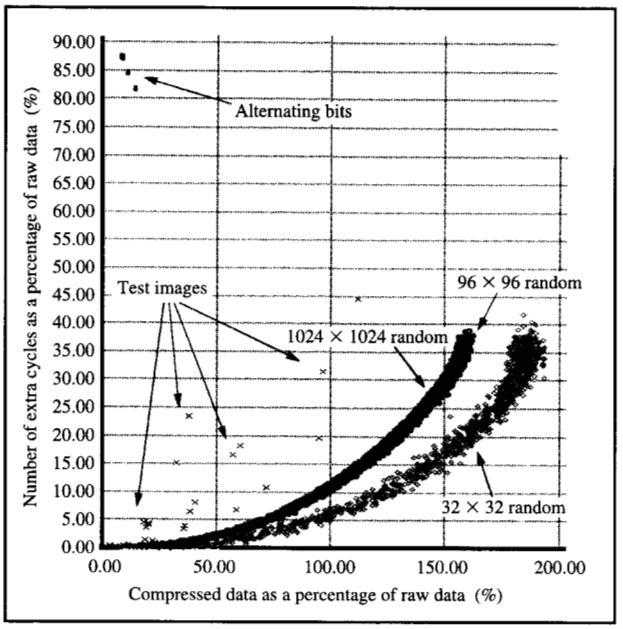

Figure 7

BLDC performance graph.

generated on the last bit of raw data. The results are displayed in Figure 6. The extra cycles include the multiple-bit renormalization penalty, the stack-clearing penalty, and, if applicable, the byte-stuffing penalty. Dashed lines have been added for reference between the sparse data points and may not portray actual performance curves. The random-image curves from Figure 5 have been added to the graph for reference.

The additional cycles incurred when clearing the stack can be considered added latency when they occur after the last bit of raw data is processed. If the stack is cleared before the raw data stream terminates, the extra cycles can be viewed as a throughput impact. Displayed in Figure 6 is the largest number of stacked bytes (35086) encountered during experimentation. Given the few data points generated, a relationship between stack counts and performance cannot be inferred.

Lempel–Ziv coders

Lempel–Ziv pattern-matching algorithms utilize a history-buffer width based upon the width of the processed data unit. Typically a byte represents the data unit; however, some implementations utilize half and full words. History-buffer lengths may also be varied in an attempt to increase the compression ratio by providing a greater range over which string matches can be located.

Inherently, implementations must buffer some compressed data because the size of a coded-data unit exceeds that of the raw-data unit. This permits more consistent generation of output data, since coded data is generated intermittently. Degradation from the maximum throughput performance can occur when coding expands and produces data at a greater rate than output can be generated.

- *ALDC throughput*

The ALDC algorithm performance can degrade only when the data expands during compression. The IBM Blue Logic core implementation of the ALDC algorithm operates upon a byte-size data unit with a history buffer length of 512, 1024, or 2048. The implementation of the ALDC algorithm avoids this bottleneck of expanding data by providing a two-byte-wide interface to generate output. Without the expansion bottleneck, ALDC compression consistently operates at the ideal throughput performance (one byte per cycle) regardless of the compression ratio obtained. Because of the lack of limiting factors, no experimentation was performed utilizing the ALDC algorithm.

Worst-case expansion of the ALDC compression algorithm is observed in data patterns that do not repeat over lengths exceeding the size of the history buffer. With a byte-size (eight bits) data unit, encoding a raw-data byte that cannot be encoded within a string match generates nine bits of coded data. When data is generally random, the number of string matches approaches 0%, and the expansion of the raw data approaches the limit of 12.5%.

- *BLDC throughput*

The BLDC compression algorithm is derived from the mating of a run-length encoding scheme and the ALDC coding algorithm. This particular approach is suited to bilevel images. The IBM Blue Logic core implementation of the BLDC algorithm interfaces with the ALDC implementation and is limited by its one-byte-per-cycle throughput. The performance of the BLDC algorithm is degraded from ideal when the run-length encoding generates data at a rate greater than one byte per cycle. This occurs with images containing many short runs.

Three sets of random images were used to characterize BLDC throughput performance. Additionally, the same set of test images used with the arithmetic coders were analyzed to determine the difference between actual images and the random images. Finally, since BLDC is sensitive to checkerboard patterns, images containing alternating bits were also analyzed. The normalized results are graphically presented as a percentage of the raw-data amount in **Figure 7**. Each point on the graph corresponds to the result of compressing these images and designates

the number of extra cycles incurred during processing in relation to the resulting amount of compressed data.

On the basis of the experimental results, images of alternating bits displayed the worst data rate. Although the compression ratio obtained was high (10:1), the data rate was 53% of maximum (0.53 bytes per cycle). Of the random images, the smallest size incurred the least penalty because of the ALDC bottleneck, when correlated to compression ratio. As size is increased, the points solidify into a well-defined line, displaying a greater degradation in throughput. As the amount of compressed data increases, decreasing the compression ratio, the number of extra processing cycles accordingly increases, and system performance is decreased as a consequence.

All of the test images maintained a higher compression ratio than the random images. However, the test images suffered more from throughput bottlenecks, consistently showing a lower data rate. The characteristics demonstrated by the experimental results indicate that realistic images, while yielding better compression than noisy random images, do not yield better performance.

The worst-case expansion of the run-length coding algorithm does not correlate to the worst-case expansion of the BLDC algorithm. The run-length and ALDC coding algorithms complement each other, and maximum expansion by the run-length encoder is compensated by reasonable compression by the ALDC encoder. This is observed in the experimental results of the alternating bit patterns. The worst-case compression that can be observed with the BLDC compression algorithm occurs when an image displays sequences of short runs that generate only small string matches when processed by the ALDC portion of the algorithm. The worst-case expansion encountered during experimentation was observed to be about 80% (compression ratio of 0.55:1). This occurred in the smallest of the random-image sizes and correlates to the worst expansion of an infinite image size, demonstrated to be around 80%.

- *ALDC latency*

As stated earlier, implementations of the Lempel–Ziv pattern-matching compression algorithms must buffer coded data to avoid a bottleneck to throughput performance. The IBM Blue Logic core implementation buffers up to two bytes before outputting coded data. When the raw data stream terminates, the ALDC coder completes the coding of the last byte and terminates the coded data with an end marker.

The maximum amount of data that may be buffered preceding the last raw-data byte is 15 bits. If coding of the last raw byte terminates a string match but cannot be included in that match, it can produce up to 31 bits of coded data (largest copy pointer + coded literal). In addition to the end marker (13 bits), the coder must potentially flush out 59 bits of coded data. Since data is processed on byte boundaries, this creates a worst-case latency of eight cycles. This is in addition to any cycles required for the last byte of raw data to traverse the internal coding pipeline.

- *BLDC latency*

Latency in the BLDC compression algorithm is a combination of the worst run-length-coding latency and the worst ALDC coding latency. The last byte of raw data may contain alternating bits, expanding in the run-length-coding portion of BLDC up to eight times. Since the ALDC coder processes only one byte per cycle, the run-length coder is throttled. This produces the worst-case performance in both throughput and latency. The eight additional cycles, when added to the ALDC worst-case latency of eight cycles, reveal that the BLDC worst-case latency can attain 16 cycles.

Summary

This paper discusses the performance of lossless data compression algorithms available as IBM Blue Logic cores. A general relationship between the compressibility of an image and the throughput performance of the compression algorithms included in the study was established. Although the experimentation was based upon hardware implementations of the compression algorithms, the results also reflect the potential of software implementations.

The use of random images in analyzing the bilevel-image arithmetic coders, ABIC and JBIG, indicates a relationship between multiple-bit renormalizations and the compressibility of an image. The ABIC bilevel-image arithmetic coder demonstrated a worst-case performance of at least 80% of the maximum data rate (0.8 bits per cycle) with a minimally compressible random image. All genuine images processed displayed better performance characteristics than the set of random images.

Although experimentation demonstrated the JBIG bilevel-image arithmetic coder to be vulnerable to diabolical images, with some expanding up to 175%, genuine images do not demonstrate the same results. Generally, the JBIG algorithm demonstrated a worst-case performance of 73% of the ideal data rate (0.73 bits per cycle) when a random image of low compressibility is encountered.

Analysis of the Lempel–Ziv string-compressing algorithm, ALDC, showed that performance is not affected by the compressibility of the raw image. Provided there are no bottlenecks in transferring data, the maximum data rate of one byte per cycle is always maintained. With run-length preprocessor extension to ALDC (BLDC), the processing of random images indicates that decreased compression ratios correlate to

decreased throughput performance. However, genuine images demonstrated an inverse relationship, producing better compression but worse performance. The diabolical checkerboard image revealed a throughput performance of 53% (0.53 bytes per cycle) while obtaining a compression ratio of 10:1.

For the most part, the latency encountered by the compression algorithms is minimal, with the exception of the JBIG compression algorithm. The use of diabolical images showed the characteristic of stacking 0xFF bytes until a carry is resolved, creating the possibility that a large amount of the compressed data may be buffered within the coder. If the potential for a carry has not been resolved before the completion of processing, time is required to clear the coded data buffered in the stack.

Acknowledgments

Many individuals contributed to the IBM Blue Logic core data compression development effort on which this study was based. A key contributor, Joan Mitchell of the IBM Thomas J. Watson Research Center, provided invaluable technical assistance during the development of the emulation software and evaluation of the experimental results. The initial code development of the Qx-coder, used as a base for further code development in this study, was performed by Michael Slattery, IBM Burlington. The continued code development would not have proceeded without the leadership demonstrated by Peter Colyer, IBM Burlington, in defining the verification requirements of the JBIG–ABIC development project. The ALDC/BLDC compression algorithms were defined by David Craft, IBM Austin. The ALDC compression code used in this study was written by Oscar Strohacker, IBM Austin, with core development assistance provided by Julie Cubino and Cory Cherichetti, IBM Burlington. Without the continued support and encouragement of my manager, Ted Lattrell, IBM Burlington, all of this work would not have been completed.

References

1. R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A Multi-Purpose VLSI Chip for Adaptive Data Compression of Bilevel Images," *IBM J. Res. Develop.* **32,** 775–795 (1988).
2. W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, Jr., and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Develop.* **32,** 717–726 (1988).
3. J. L. Mitchell and W. B. Pennebaker, "Software Implementations of the Q-Coder," *IBM J. Res. Develop.* **32,** 753–774 (1988).
4. "Information Technology—Coded Representation of Picture and Audio Information—Progressive Bi-Level Image Compression," *ITU-T Rec. T.82\|ISO/IEC 11544: 1993*, JBIG standard, 1993.
5. W. B. Pennebaker and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, ISBN 0-442-01272-1, 1993.
6. K. M. Marks, "A JBIG–ABIC Compression Engine for Digital Document Processing," *IBM J. Res. Develop.* **42,** 753–758 (1998, this issue).
7. M. J. Slattery and J. L. Mitchell, "The Qx-Coder," *IBM J. Res. Develop.* **42,** 767–784 (1998, this issue).
8. D. J. Craft, "ALDC and a Pre-Processor Extension, BLDC, Provide Ultra-Fast Compression for General Purpose and Bit Mapped Image Data," *Proceedings of the DCC'95 Data Compression Conference*, 1995, p. 440.
9. D. J. Craft, "A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions," *IBM J. Res. Develop.* **42,** 733–745 (1998, this issue).
10. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Info. Theory* **IT-23,** 337–343 (1977).
11. W. B. Pennebaker and J. L. Mitchell, "Probability Estimation for the Q-Coder," *IBM J. Res. Develop.* **32,** 737–752 (1988).

Received February 4, 1998; accepted for publication July 14, 1998

Francis A. Kampf *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (kampf@btv.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 cores.