HOW LZS DATA COMPRESSION WORKS

Application Note

hifn

Hi/fn™ supplies two of the Internet's most important raw materials: compression and encryption. Hi/fn is also the world's first company to put both on a single chip, creating a processor that performs compression and encryption at a faster speed than a conventional CPU alone could handle, and for much less than the cost of a Pentium or comparable processor.

As of October 1, 1998, our address is:

Hi/fn, Inc.
750 University Avenue
Los Gatos, CA 95032
info@hifn.com
http://www.hifn.com
Tel: 408-399-3500
Fax: 408-399-3501

Hi/fn Applications Support Hotline:
408-399-3544

Table of Contents

Figures

THIS PAGE INTENTIONALLY BLANK

1 Overview

In storage technology it is desirable to have greater storage capacity at lower costs. Data compression addresses these demands by reducing the amount of data that must be stored to a given size of media, thus lowering the cost of that storage device.

In data communications it is desirable to have faster transfer rates at lower costs. Data compression addresses these demands by reducing the amount of data that must be transferred over a media with a fixed bandwidth, thus reducing the connection time. Data compression also reduces the media bandwidth required to transfer a fixed amount of data with a fixed quality of service, thus reducing the tariff on this service.

Data compression works by having a compressor at one end of the data link and a decompressor at the other end. For full-duplex communication there would be a compressor for the forward channel and a decompressor for the reverse channel at each end of the communication link.

2 How LZS Data Compression Works

The *LZS* compression algorithm works by searching for redundant data strings in an input data stream and replacing these strings with encoded *tokens* of shorter *length* in the output data stream. The LZS algorithm builds a table of these string matches that consists of pointers into the previous data from the input stream. The encoded tokens that are used to replace redundant strings are created from information within this table. In this way future data is compressed based on previous data.

Thus, the more redundant the data in the input stream the higher the *compression ratio* will be, and conversely, the more random the data in the stream the lower the compression ratio will be.

The compression engine maintains a compression history, which contains the last 2Kbytes of input data, as well as other data structures to accelerate the compression operation. The compression engine uses this history to search for string matches to create the tokens. Similarly, the decompression engine at the other end of the data link maintains an identical decompression history, which contains the last 2KBytes of output data. The last 2Kbytes of input data maintained in the compression history must be identical to the last 2Kbytes of output data maintained in the decompression history. The decompression engine uses its history to recreate the raw data from the tokens provided by the compression engine. The compression and decompression histories at each end of the data link must always match, otherwise the *decompressor* may output garbage pointed to by the token.

The compressed stream data consists of *literal data* and *compressed tokens*. Literal data are input data strings that could not be matched (compressed). Compressed tokens consist of an *offset* to a location in history that contains the string match, and a *length* of number of bytes that match this string. An example would be “go back 150 bytes and output the next 10 bytes”. Thus, the tokens can be viewed as pointers into the 2 Kbyte compression history

The compression ratio is a result of two key factors: the actual data being compressed, and the amount of searching done to match the "best" (longest) string. Thus compression consists of intensive string searching and matching. Many LZS products have a "tunability" feature where compression ratio can be traded for compression speed by changing the amount of searching performed. Decompression is much simpler, and hence much faster algorithm. The LZS decompression algorithm works by taking the compressed data and performing one look-up for each compressed token. The decompressor reads the *offset*, jumps to that location and outputs the *length* of bytes starting at that offset. Thus decompression is much faster operation than compression, and does not require the tunability feature.

At the end of each compression operation, the LZS engine must be *flushed*. Flushing consists of outputting any data held inside the LZS engine, and outputting an *end marker*. Data is held inside the LZS engine because the engine may be in the middle of a long string match. A *flush* operation terminates this string match process. An end marker is a unique token (used by the decompressor to find the end of the compressed data).

2.1 Maintaining the Compression History

In some applications data is compressed in small amounts. If the history information regarding the input stream were reset after each compression operation, the resulting compression ratio would not be optimum. Many of Hi/fn's data compression products allow the history information to be maintained between multiple compression operations. This allows each successive compression operation to utilize history information from the previous compression operations. This yields a higher compression ratio when compressing small amounts of data.

For example, in data communications data is transmitted in small units called packets. These packets may be quite small (e.g. tens of bytes). The LZS compression engine must be flushed for each *packet* so the compressed data can be transmitted and the next packet may be processed. If the history were to be reset on each packet, the overall compression ratio would be poor. However, if the history information is maintained between flushes for each packet, then the result would be a significantly higher compression ratio.

A property of Hi/fn's LZS algorithm is that maintaining or clearing the history is a function entirely of the *compressor*. Under normal operating conditions the decompressor doesn't know and doesn't care if the compressor maintained or cleared the history between successive packets. That is, the compressed data is "self-extracting".

Referring to Figure 1, if you consider the LZS encoded format of offsets and lengths, maintaining the history implies that there may be offsets pointing into previous packets that have been compressed in this history (up to 2Kbytes back). Clearing the history between packets implies that there will be no offsets pointing prior to the beginning of the current packet. Either way, the decompressor still reads the *offset* and outputs the number of bytes specified by the length.

Thus, it is important to note that when keeping history that the decompressor must decompress packets in the same order they were compressed in. Again, if

successive packets are decompressed out of order, then an offset may point to the wrong data string and the resultant output will appear to be garbage. Conversely, when clearing the history after each packet it does not matter which order packets are decompressed in a given history.

```
<Compressed Stream> := [<Compressed String>] <End Marker>
   <Compressed String> := 0 <Raw Byte> | 1 <Compressed Bytes>
   <Raw Byte> := <b><b><b><b><b><b><b><b>          (8-bit byte)
   <Compressed Bytes> := <Offset> <Length>

   <Offset> := 1 <b><b><b><b><b><b><b> |           (7-bit offset)
               0 <b><b><b><b><b><b><b><b><b><b><b> (11-bit offset)
   <End Marker> := 110000000

<b> := 1 | 0

   <Length> :=
   00         = 2     1111 0110      = 14
   01         = 3     1111 0111      = 15
   10         = 4     1111 1000      = 16
   1100       = 5     1111 1001      = 17
   1101       = 6     1111 1010      = 18
   1110       = 7     1111 1011      = 19
   1111 0000 = 8      1111 1100      = 20
   1111 0001 = 9      1111 1101      = 21
   1111 0010 = 10     1111 1110      = 22
   1111 0011 = 11     1111 1111 0000 = 23
   1111 0100 = 12     1111 1111 0001 = 24
   1111 0101 = 13      ...
```

Figure 1. LZS data compression format

2.2 Multiple Histories

Bridges and routers are classical examples of applications that process multiple streams of data concurrently. A router may establish multiple virtual connections and transfer independent data streams over each virtual connection. When compressing multiple streams of data it is important to note that data from the same source and applications and bound for same destinations tends to be redundant, but data from different sources and applications bound for different destinations tends to be more random. That is, statistically the data in one particular data stream typically is redundant. Conversely, the data in different streams (which may represent separate virtual connections) will be less redundant.

For example, one data stream can be an executable program loading for one user, while another data stream can be a word processor file being down-loaded to another user, while a third data stream could be a spreadsheet database loading from a third user. Each data stream probably has redundant, compressible data in it. However, each of the 3 channels probably have vastly contrasting data from the others.

The problem is that a router cannot wait to receive the entire file before compressing and transmitting it. This would require tremendous amounts of memory and create tremendous amounts of delay (called *latency*) inside the router. So, the router chops up the data stream into manageable sized packets of information and transmits these packets on all 3 data streams concurrently.

Since data is transferred in small packets, the router is constantly switching between virtual connections to minimize *latency* for each session. However, unless care is taken, this can reduce the compression ratio of data as it passes through the router.

One option is to utilize one history and clear the history information after each compression operation when the router switches from one *virtual connection* to another. As seen in the previous section this is undesirable, because there will never be much of a history to search for matches, and the compression ratio will suffer.

Another option is to utilize one history and maintain all information in all data streams in this one history. If the data streams are transmitted to multiple points, then this technique will fail miserably as data bound for different destinations are compressed in this one history, yet these data streams are decompressed in histories that may not be synchronized to this history at all. Even if all the data streams are received at the same destination, the compression ratio will be poor unless all data streams are passing the same kind of data.

To achieve maximum redundancy over multiple data streams, and therefore higher compression ratios in each data stream, the router needs to maintain separate histories for each data stream, and efficiently switch between these multiple histories.

While Hi/fn's research has determined that 2K bytes is the optimal history size for the LZS algorithm, typical data communication packets are much smaller. Since data is transferred in small packets, the router is constantly switching between virtual connections (and their associated histories) to minimize latency for each session and maximize compression ratio in each data stream.

Maintaining multiple histories requires that history data consisting of the input data stream, as well as certain data structures maintaining string matches into the history, and the state of the compression engine be saved. The compression and decompression engines will maintain pointers into the compression history memory area for each data stream. In this way, the compression engine may be switched efficiently between the various data communication sessions with their individual data compression histories. These data structures as well as their histories are allocated separately for each virtual connection.

The cost of this efficiency is the amount of memory in the compressor and decompressor, which is typically in the 16KB range for a full-duplex communication stream.

Just as it is important to maintain history information between successive compression operations over a single history, it is equally important to maintain history information over multiple histories.

Thus, the concept of keeping history (and conversely clearing history) needs to be implemented in products that implement multiple histories.

2.3 Data Expansion

There are some kinds of data that will not compress. That is, it is possible for the compressed data to be larger in size than the raw data. This is called data expansion.

If the application is data communications where the goal is to preserve bandwidth on a wide area data link, then there are 3 possible solutions. The first option is to simply transmit the expanded data and suffer the lost bandwidth in the current transmission. This preserves the potential compression ratio for future data transmissions as the history may be maintained. The good news is that LZS can only expand to a maximum of 12.5% under worst case conditions, that is the raw data is totally random. In this case the transmission speed would be 12.5% slower than sending the raw data. Also, as seen below, this simplifies the *transmitter* and *receiver* implementations.

However, in some systems this lost bandwidth is unacceptable. And it is possible that subsequent packets may not compress also. The second option to deal with data expansion is to send the raw data and reset the compression history. This technique preserves the bandwidth of the current data transmission, but causes future data transmissions to suffer because the *compression history* will need to be rebuilt.

This second option needs special handling by the transmitter. The *protocol* layer of the transmitter must notify the receiver that this packet of data is not compressed, and the receiver must not decompress this packet of data. Since the compressor already compressed the data for this packet even though it did not transmit it, the decompressor's history is no longer identical to the compressor's history, and the compressor's history must be cleared.

The third option is to send the raw data and update the decompressor's history with the raw data. This preserves the bandwidth of both the current data transmission and future data transmissions. This will also resynchronize the compressor's and decompressor's histories while maintaining the highest compression ratio. Again, the *protocol* layer needs to inform the receiver not to decompress this data packet, but to update the decompressor's history with the raw data packet.

Hi/fn refers to this feature as "Anti-Expansion".

3 Summary

Data compression is a ubiquitous technology, because it provides high value for its cost. It is important that the system implementor understand how this technology works so its application can be utilized most efficiently. It is also important for the system implementor to understand some basic principles of the application of data compression to the system. Maintaining compression histories, associating compression histories with virtual connections, and implementing anti-expansion mechanisms are vital to successfully implementing an optimal compression system in a router.

4 Appendix 1: A list of Hi/fn LZS Standards

Hi/fn's LZS algorithm has been standardized, and information is available from the following institutions:

ANSI X3.241-1994: American National Standards; 11 West 42nd St., New York, NY, 10036 (212) 642-4900

QIC-122: Quarter-Inch Cartridge Drive Standards, Inc.; 31 East Carrillo St., Santa Barbara, CA, 93101, (805) 963-3853

IETF RFC1974: Internet Engineering Task Force; http://info.internet.isi.edu/in-notes/rfc/files/rfc1974.txt

FRF.9: The Frame Relay Forum; 303 Vintage Park Drive, Foster City, CA 94404-1138 (415) 578-6980, frf@sbexpos.com

TIA/EIA 655: Global Engineering Documents; 15 Inverness Way East, Englewood, CO 80112-5704 (800) 854-7179

5 Appendix 2: Hi/fn's LZS Encoding Format

Figure 1 on page 7 shows the format of Hi/fn's LZS compressed data format, which is also defined in many standards, such as ANSI X3.241-1994.

By inspecting the length field of the LZS format, it can be seen that the maximum compression ratio is 30:1. That is, for every 15 bytes in the input stream that are matched, a nybble of 0xF is output. Thus for every 30 bytes input, there is one byte output.
Conversely, it can be seen that the worst case expansion is if every byte is raw, so 9 bits are sent to the output stream for every 8 bits in the input stream.

The LZS end mark is a compressed token consisting of a unique 7-bit offset of 0x0.

6 Appendix 3: Glossary

Compression - A function to decrease the amount of data in a system. In a networking application this is a function of the transmitter to increase the bandwidth of the data channel. This function can be implemented in hardware or software.

Compression ratio - The input number of bytes divided by the output number of bytes

Compressed tokens - In the LZS algorithm, output tokens that represent compressed input data that matched in the history. Compressed tokens consist of offsets and lengths.

Compression history - In the LZS algorithm, the memory reserved to store the last 2Kbytes of input data and implement the compression algorithm.

Data channel - A physical connection between two communication devices.

Data stream - Data that flows from a transmitter to a receiver via the data channel.

Decompression - A function to increase the amount of data in a system. In a networking application this is a function of the receiver to restore the bandwidth of the data channel. This function can be implemented in hardware or software.

Decompression history - In the LZS algorithm, the memory reserved to store the last 2Kbytes of output data and implement the decompression algorithm.

End marker - a unique 7-bit token produced during the flush operation, which the decompressor uses to find the end of the compressed data.

Flush - to terminate a compression operation, the compressor's engine must be flushed of any data being cached for string matching. Also, the flush operation appends an end marker to the compressed data so the decompressor can find the end of the compressed data.

Latency - the amount of time delay from when a router receives the last byte of a network packet until the last byte of the WAN datagram is transmitted.

Length - In the LZS algorithm, the number of input data bytes that matched.

Literal data - In the LZS algorithm, compressed data that did not match in the history and would not compress.

LZS - Hi/fn's lossless data compression algorithm.

Offset - In the LZS algorithm, the distance from the current data that the matched compressed string starts.

Packet - In a networking application, a unit of data transmitted to the receiver. For example, ethernet packets are 1500 bytes maximum.

Protocol - A formal set of rules governing the format, timing, sequencing, and error control of exchanges messages on a data network.

Receiver - In a networking application, the receiver of data via the data channel during the data transfer.

Tokens - In the LZS algorithm, the compressed data consisting of offsets and lengths.

Transmitter - In a networking application, the sender of data via the data channel during the data transfer.

Virtual connection - In a networking application, the logical connection maintained by the transmitter and receiver.

7 Appendix 4: Evaluation Software

Hi/fn makes available a demonstration program for Hi/fn's customers to try data compression on their data. The effects of maintaining history, multiple

histories, and data expansion with the LZS algorithm may be observed with this evaluation software.

This program is called CDEMO.EXE and is available on Hi/fn's world-wide web site at www.hifn.com.