Compression Techniques for AIDA

As the amount of data collected by the front-end electronics of modern digital data acquisition systems increases, the speed with which it can be transferred to the processing components becomes a limiting factor. A problem arises if more data is generated that the available bandwidth can handle: data will be lost or deadtime increased.

There has been a considerable investment, both academic and commercial, into compressing data. One of the main driving forces has been the need to utilise network bandwidth more efficiently when streaming audio and video data. Another requirement has been the need to store archive files in as small a space as possible on backup media. These two requirements sometimes overlap and sometimes conflict.

As a general rule it can be said that faster compression algorithms compress data less well than slower ones. This, in turn, means there is a trade-off to be made between different methods, when choosing a technique for streaming or other on-the-fly applications.

It has also been found that some types of data are more amenable to compression than others and that, in general, different data types can be better compressed by using an algorithm that has been tuned to the type. That is to say, prior knowledge of the data structure can aid the design of the algorithm.

The aim of this investigation was to look at the efficiency of different compression methods when applied to some typical detector data streams. The technique must be a form of Lossless Data Compression[i], which allows the original data to be exactly recovered.

Well-known Techniques

A brief Internet search found a number of compression methods. In addition, 2 techniques were found that have been implemented in VHDL for use in an FPGA.

- Rice[ii] – a form of entropy encoding (essentially a bin number & remainder, encoded)
- Huffman[iii] – frequency analysis & table based on probability of occurrence of pattern
- LZ77[iv] – matching strings & storing in dictionary
- LZO[v] – similar to LZ77, open source aimed at real-time apps
- Zlib[vi] – as used by gzip, combines LZ77 & Huffman. Commonly available
- RLE[vii] – run length encoding
- LZRW3[viii] – commercial IP for Xilinx of a patent-free public domain algorithm
- X-Match PRO[ix] – commercial, for chip implementation, Partial Pattern Matching.
  (This was harder to test. A Windows example/demo program was provided, and smaller sample used.)

The LZ77, LZO and RLE encoding implementations gave poor results in terms of compressed file size and are not shown in the results.

Some other variations were found but were also found to be very slow or inefficient. Although it was noticeable that the algorithms' performance was often quite implementation dependent, these variants were not looked at further.

Bespoke Techniques

There are many methods that are designed for a particular type of data, e.g. audio, video or text. It was decided to implement some methods based on knowledge of the data structure of the ADC data stream and the requirements for the compressed data.

- The data consists mainly of noise, interspersed with peaks from a hit on the detector.
- The ADCs producing the data are 14 bits wide.
- It is not necessary to compress the peaks, which are comparatively sparse.
- The noise should be at a low level, even though the signal level may not be.
- The data is to be compressed from fixed-length source data blocks. The blocks are 2000 16-bit words long.

The aim was to produce a simple method that reduces the amount of transmitted data per word without reducing its information content. The method should compress typical data (noise) well. Failure to compress atypical data (peaks) is not a problem. It was hoped to produce compressed data files that could be "read by eye" without too much difficulty in order to help verify their correctness.

Method 1. Minimum & Offset

The data is low noise. In most cases it will exhibit good value locality. This means that if the data is represented by the offset from a minimum or base value, the range of offsets will not be large. This, in turn, means that the offsets can be held in byte or even 4-bit units. The data block can be compressed to a 16-bit word for the base value and 2000 8-bit values for the offsets, or a 16-bit word for the base value and 2000 4-bit values for the offsets (1000 bytes). The method is two-pass. The first pass is to scan the data for its minimum and maximum values in order to calculate the range – which must not exceed 256, the maximum that can be held in a single byte. The second pass calculates the excursion from the minimum for each data word and writes it to the output buffer. If the range is no greater than 16 the output data can be held two words to a byte. These two variants will produce a compression ratio of nearly 2:1 or 4:1 for suitable source data.

Method 2. Dictionary Lookup

Although the data is low noise there may be spikes. However, since it is noise it is not likely to display ramped changes, except during the decay of a peak, so the number of distinct values in a block should not be very large. The effect should be that it might be possible to store each data word as an index into a dictionary of words, in one byte. The data can be represented as a set of 16-bit words for the dictionary and 2000 8-bit values for the indexes into it. The method is two-pass. The first pass is to scan the data for its distinct values in order to create the dictionary – the size of which must not exceed 256, the maximum that can be held in a single byte. The second pass

calculates the index of each data word and writes it to the output buffer. This method produces variable length output blocks depending on the number of distinct values. The best compression, for 1 value is the same as the previous method. Usually the Minimum & Offset method will produce better results but if the data contains a few words with very different values the Dictionary Lookup method will be successful.

Limitations
In some cases the data in a block will not be amenable to either of these methods and must be output raw.

Optimisation – Bit-packing
The methods described above can be optimised and extended by including an element of Zero-Awareness. This is achieved by bit-packing which is, effectively, a generalisation of the 2-words-per-byte method described above. If, for example, the maximum offset (or index) is 50 it can be represented by 6 bits and there will be 2 wasted zero bits in each byte transmitted. It would be more efficient to pack the offsets 6 bits at a time; 4 6-bit words would be packed into 3 8-bit bytes. Bit-packing also allows for the possibility of going beyond 8 bits. In the normal case, if the range or dictionary length is greater than 256, there is no advantage in encoding the offsets as 16-bits since the original data itself consisted of 16-bit words. However, if the redundant zero bits can be discarded by bit-packing a significant advantage can be gained. If, for example, the maximum offset (or index) is 500 it can be represented by 9 bits. It would be more efficient to pack the offsets 9 bits at a time; 8 9-bit words would be packed into 9 8-bit bytes, which is much better than 8 16-bit words. In fact, since the data is from a 14-bit ADC, even when neither Minimum & Offset nor Dictionary Lookup results in a reduction in size, bit-packing will reduce the data size from 4000 bytes to 3500 bytes.

The Test Program

Subroutines were written or incorporated that all implemented all the algorithms under test. They were then linked into a test program that read data in and compressed it with the method chosen on the command line.
e.g. `compressData –c huff –f Seg2.dat`

The data was read in from the files in 2000 word blocks. Each block was compressed and written out to an output file. Thus each output file contained all the data from its input file but had been compressed 2000 words at a time. No information was held, within the program, from one iteration of the compressor to another.

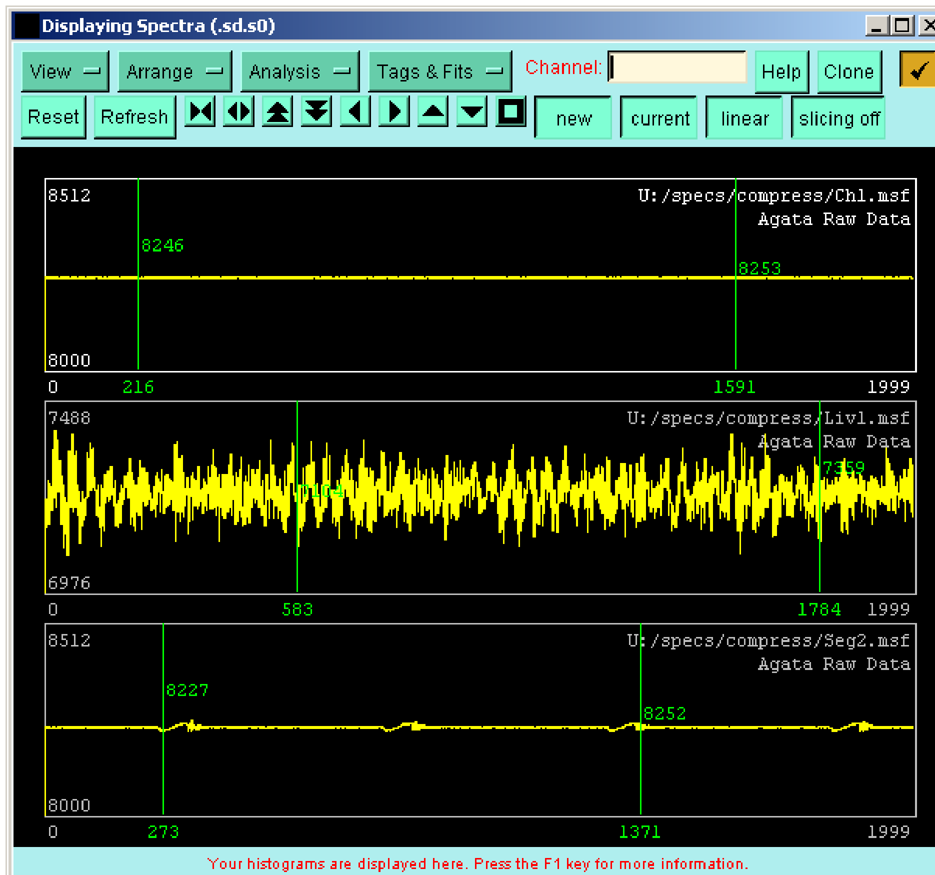
The program included the bespoke Minimum & Offset and Dictionary Lookup methods, both with and without bit-packing. It also included an “optimum” method that, following a first-pass through the data, makes a decision as to which method will provide the best result.

Data

8 sample data files were used. All were 128MBytes, i.e. 32000 blocks, long.

2 were produced by a test setup in the laboratory at Daresbury. There was no detector, so these files simply contained a low noise level. In the results, these are known as Ch1 and Ch2.

6 came from Liverpool. The Liverpool files were in 3 sets of 2 pairs. Of each pair, one was very noisy, the other was less noisy but contained a noticeable pulse every 512 channels. They also contained detector data.



In the results, these are known as Liv1 & 2, Scr1 & 2 and Seg1 & 2.

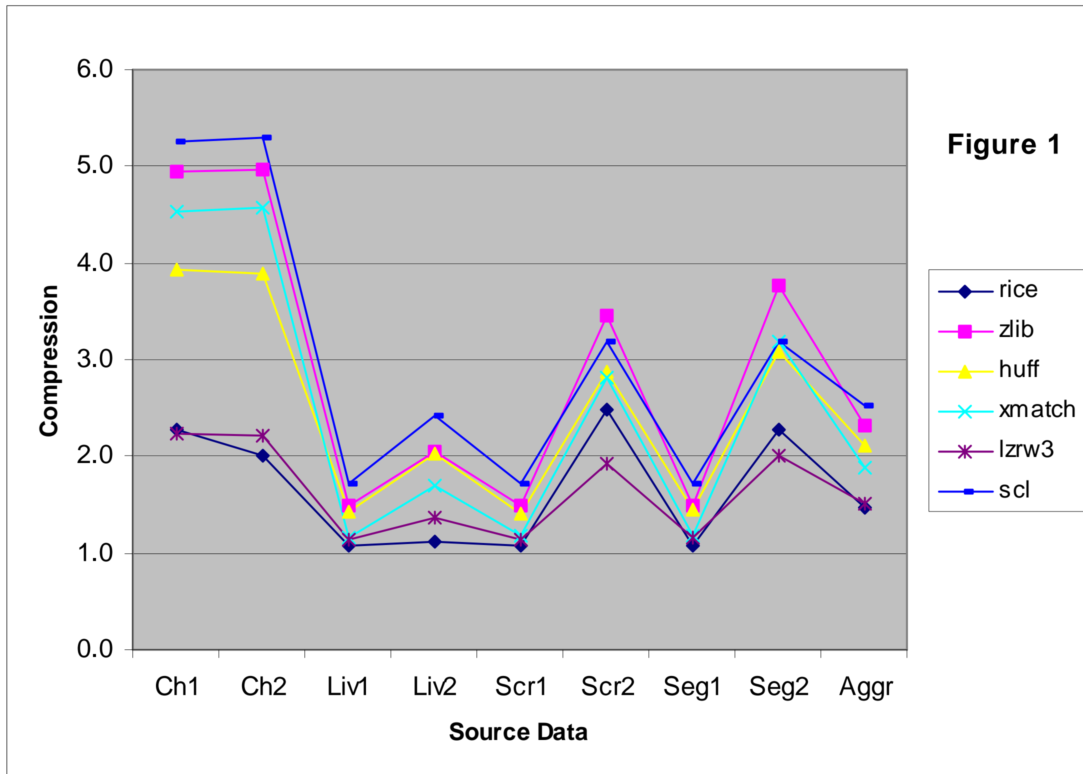
Results

Compression Ratios

The compression results are expressed as the original file size (128MB) divided by the results file size. When looking at the compressions values bigger is better.

The results shown in Figure 1 indicate that to a greater or lesser extent the sample data is compressible in the same way. That is, if it compresses well with one technique it will also do so with another.

The same Figure shows that although the results do vary from method to method the bespoke optimum method produces compression that is often better than the well-known zlib library.



Several implementations of Huffman encoding were tried which gave rather different results, indicating that further investigation of that method might be worthwhile. However, it was decided to look at the bespoke methods in greater depth.

Figure 2 shows 2 particular results.

1. The effect of bit-packing.
2. The effect of choosing the method, compared with only using one method for all blocks.

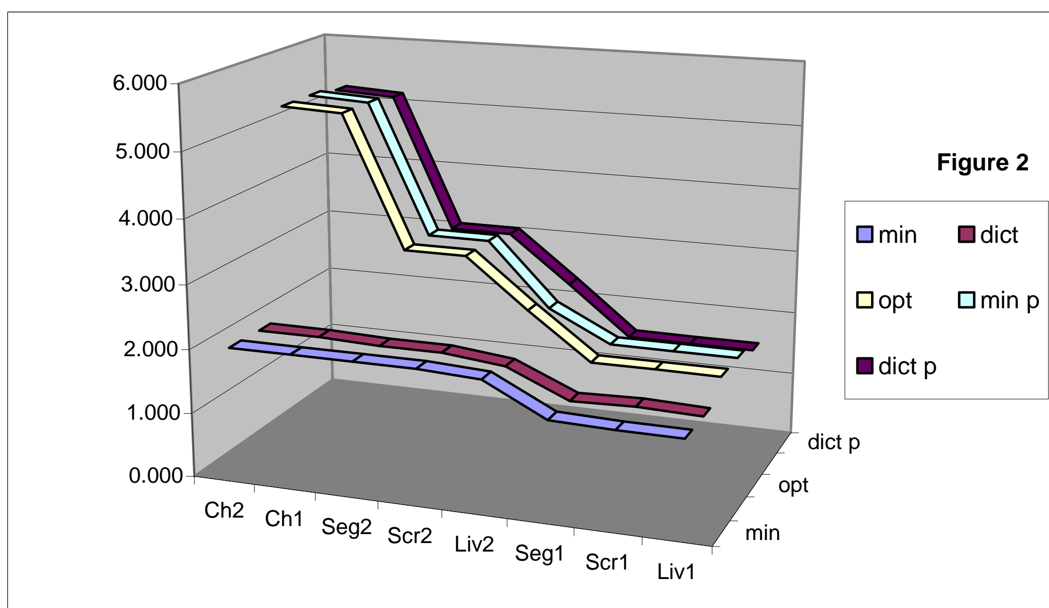
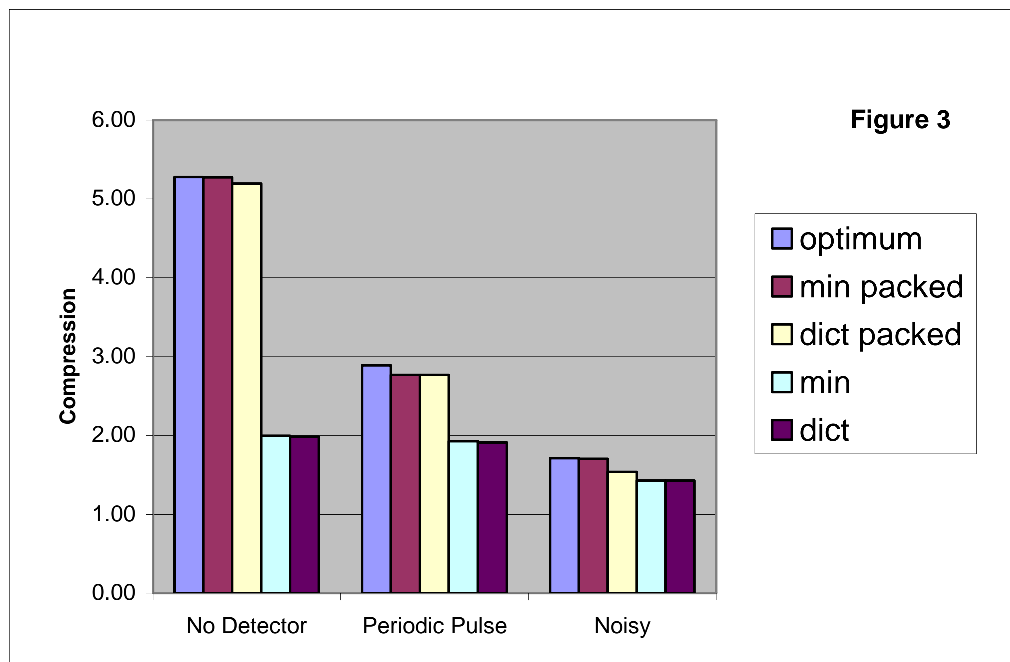


Figure 3 puts the results together by data type. None of the methods work well if the data is not very compressible but the packed methods are significantly better for compressible data.



It can be seen that the best results are obtained by bit-packing one of the other methods and that Minimum & Offset, which sometimes gives poor compression, if any, on its own usually gives ratios near the optimum when packed. The Data Dictionary approach can also give the good performance but not usually the best.

Speed
Since the intention is to implement the chosen method in VHDL in part of a logic array a simple method is likely to be a good choice.

The speed of operation is highly implementation dependent and therefore direct speed measurements are not directly relevant to this investigation. Nevertheless speed is, probably, an indication of the complexity of the algorithm so the speed of compressing the sample data blocks was measured.

The X-Match PRO test program reported it was compressing at 200MB/s but actually was much slower. This was probably the speed it claimed to be able to achieve from a hardware implementation. This could not be directly compared with the other methods, which were run on a different machine and Operating System.
LZRW3 was fast but did not achieve particularly good compression ratios.
Huffman was quite fast.
Zlib generally gave the best compression but was always the slowest method.
The simple bespoke algorithms, Minimum & Offset and Dictionary Lookup, were fast but only achieved good results on a small subset of the data. Neither coped very well with noise. Bit-packing slowed these methods down considerably.

Conclusions

The bespoke algorithms are conceptually simple and probably suitable for a hardware implementation.

It is necessary for the method to choose between compressed and raw, non-compressed, output. This is because, sometimes, the compressors produce out put that is larger than the source data.

When bit-packing is introduced they produce good results at the expense of speed. It is likely that this speed reduction is due to the difficulty of implementing what is essentially a bit-oriented technique on a byte- or word-oriented machine. The program has to do a lot of shifting and masking in order to put the bits into the correct places in the output stream of bytes. It is hoped that a hardware implementation would not suffer from this since the bits could be acted on directly.

Both algorithms are two-pass. It is possible that the first pass could "come for free" in a hardware implementation if the required information could be extracted from the data stream as it passed on its way to the compressor.

Although the Dictionary method can improve results in some cases, the added complexity of implementing a second algorithm may not be worthwhile.

The most suitable approach would seem to be bit-packed Minimum-or-Raw.

Appendix

C-code fragment to demonstrate recommended method

```
#define INPUT_LENGTH 2000
#define RAWP_LENGTH ( (sizeof(short) + ((INPUT_LENGTH*14) / 8)) )

#define BitWrite(Out,Index,bitPos,bitSet) { \
	if (bitSet) Out[Index] |= (1 << bitPos); \
	bitPos = (bitPos + 1) % 8; \
	if (!bitPos) Out[++Index] = 0; \
}

int MinP_Compress(void *inD, void *outD)
{
	int i, j, outLen, NumBits, bitPos;
	unsigned short *in = (unsigned short *)inD, d;
	unsigned char *out = (unsigned char *)outD;
	unsigned short minV, maxV;

// Find the Maximum and Minimum values in the block
	minV = maxV = d = in[0];
	for(i=1; i < INPUT_LENGTH; i++) {
		d = in[i];
		if (maxV < d) maxV = d;
		else if (minV > d) minV = d;
	}
// Calculate range from Max & Min
	i = maxV - minV;
// Find min number of bits needed to hold range
	for (NumBits = 1; NumBits < 16; NumBits++) {
```

```
            i >>= 1;
            if (!i) break;
        }
// Calculate output block size to hold data packed to this bit-length
        outLen = sizeof(short) + (IN_LEN * NumBits / 8) +
sizeof(short);
// If no compression output as is, packed
        if ( RAWP_LENGTH < outLen) {
                return RawP_Compress(inD, outD);
        }

// Output flag to describe compression type & number of bits
        *(unsigned short *) (out) = (MIN_METHOD << 8) + NumBits;
        outLen = sizeof(unsigned short);
        out[outLen] = 0;
// Loop round whole block
        bitPos = 0;
        for(i=0; i < INPUT_LENGTH; i++)
        {
// Calculate offset for data word
            d = in[i] - minV;
// Pack word into correct number of bits
            for (j=0; j<NumBits; j++) {
                BitWrite(out, outLen, bitPos, d & (1 << j));
            }
        }
// Output minimum value
        *(unsigned short *) (out + outLen) = minV;
        outLen += sizeof(unsigned short);

        return outLen;
}
```

Simon Letts, STFC Daresbury Laboratory, August 2008

---

[i] http://en.wikipedia.org/wiki/Lossless_data_compression
[ii] http://bcl.comli.eu/
[iii] http://en.wikipedia.org/wiki/Huffman_coding
[iv] http://en.wikipedia.org/wiki/LZ77
[v] http://www.oberhumer.com/opensource/lzo/
[vi] http://zlib.net/
[vii] http://en.wikipedia.org/wiki/Run-length_encoding
[viii] http://www.heliontech.com/compression.htm, http://www.ross.net/compression/lzrw3.html
[ix] http://www.lboro.ac.uk/departments/el/research/esd/projects/web_xmatch/XMatchPRO_home.htm