LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-407448

Design and Implementation of an In#Cache Archival Entropy Coder for the Sonoma Persistent Surveillance System

J. G. Senecal

October 2, 2008

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Design and Implementation of an In–Cache Archival Entropy Coder for the Sonoma Persistent Surveillance System

Joshua G. Senecal
Lawrence Livermore National Laboratory
senecal1@llnl.gov

October 2, 2008

Abstract

The Sonoma sensor, developed at Lawrence Livermore National Laboratory, is an electro–optical (EO) sensor composed of a tiled arrangement of cameras, where each camera sends 20 megabytes (MB) of raw data per frame. Depending on the sensor configuration, each sensor image, in aggregate, can total up to 320 MB. I discuss here the design and implementation of a high–speed entropy coder utilizing the Lead–1 encoding method, intended to quickly and losslessly compress sensor data before it is written to storage. More sophisticated (and slower) compression techniques may be performed off–line. The coder, favoring speed over coding efficiency, is simple in principle and tiny in practice: the amount of allocated memory required by the coder can be as small as 26 bytes. On a 2.7 GHz PowerMac G5 this coder is capable of encoding sensor data at a rate of 163 megabytes per second. The overall compression rate is better than that of BZIP2, GZIP, and a range coder, and the total execution speed is superior.

1 Introduction

The Sonoma Persistent Surveillance System consists in part of an EO sensor composed of a tiled arrangement of 16 cameras, where each camera image is 20 MB in size, and therefore each sensor image is 320 MB in size. The sensor is typically triggered at 2 Hz (one frame every 500 ms), yielding a throughput of 640 MB/sec. Currently, for research purposes all sensor data needs to be stored without loss. If compression is used, it must take place in less than 500 ms.

In this report I describe the design of an entropy coder implemented for the purpose of archiving individual camera feeds in real–time. The coder is designed on the premise that for greatest speed the encoding process should consist only of simple operations that as much as possible map to hardware instructions (additions, bit shifts, etc.), and all information required for encoding (variables, tables, etc.) should reside as close to the CPU as possible, and preferably on it. The number of symbols to encode is reduced by a method called Lead–1 encoding; this results in a reduction in memory required for storing code tables. By further leveraging a single hardware instruction present on the PowerPC architecture the total amount of memory required can be reduced to 26 bytes.

Most sophisticated entropy coding techniques are comparatively slow. That is, their execution time is not always adequate for quickly encoding large amounts of data[1]. Certain coders, such as arithmetic [4, 2, 10] and range coders [5], recompute their codes fairly frequently, sometimes as much as once per symbol encoded.

For speed the coder described here uses static codes, recomputed infrequently and provided via table–lookup. The rationale for using static code tables is twofold: First, although current CPU speeds allow many things to be obtained by computation faster than by table lookup, my own tests indicate that entropy codes do not necessarily fall into this category. Second, in persistent surveillance applications the system is taking many images of what is essentially the same scene. It is reasonable to assume that the distribution of pixel values will not differ much between frames, and that a set of code tables generated with one frame will likely be effective with another.

I choose to use canonical codes. Canonical codes are equivalent to Huffman codes in terms of coding efficiency, but methods exist to create them quickly and with a minimal use of resources. The coder de-

[1] This is the driving reason behind the development of modern hardware–based compression.

scribed here generates codes according to the method presented in [6]. Although Canonical/Huffman codes have a lower coding efficiency when compared to other coding techniques, their simplicity allows them to be used in a very optimized fashion.

2 Prior Work

There is extensive prior work in developing entropy coders for various purposes, so much so that it is impossible to present a complete review here. As it relates to wide–area surveillance, experience is that most methods currently involve using sophisticated industry–standard methods (e.g. JPEG 2000, MPEG4, etc.) and getting them to operate in real–time, frequently by leveraging dedicated, custom hardware.

The work of Burtscher et al [1] is related to the coder described here. Burtscher's work involves the use of Lead–1 encoding to rapidly encode double–precision floating–point data as it is generated by a scientific computation. At the heart of the method is the use of a value predictor, whose design is based on results of research into hardware routines developed to speed program execution by predicting computation outcomes. In their work a value predictor is used to predict a floating–point value, and the XOR is taken of the predicted and actual values. If the prediction is reasonably close, the leading bits of the coefficient are zero. The number of leading zeros are counted and stored, and everything following the leading zeros is passed along unchanged. On their test data sets they report average higher compression rates than coders such as BZIP2 and GZIP, and encoding speed 8 to 300 times faster.

In their work there is no explicit use of an entropy code (although it is certainly possible to do so). For speed they instead use a fixed-length counter to store the number of leading zeros in each coefficient. The effectiveness of the method rests on the effectiveness of the predictor used.

3 The Problem Size Problem

The coder described here uses table lookups. The weakness in encoding and decoding via table–lookup is the method's sensitivity to problem size: a single table entry is needed for every symbol appearing in the data being compressed. If n–bit numbers are being encoded it should be obvious that the encoding table will require $N = 2^n$ entries. With Canonical/Huffman codes, in the worst case the longest code will be $L = 2^n - 1$ bits in length. In this extreme case, encoding 8–bit values can require a decoding table with 2^{255} entries. In practice it is extremely rare to have such a case, but even smaller values of L, such as $L = 15$ can make decoding tables undesirably large.

In the case of Sonoma, each pixel is 16 bits wide, containing 12 bits of actual data. There are potentially $2^{12} = 4096$ different symbols. The decoding table may potentially require 2^{4095} entries in the worst case. To help ensure a fast execution time table sizes must be smaller, and to guarantee that the code tables are small the problem size must be reduced. This is done via Lead–1 Encoding.

4 Lead–1 Encoding

Lead–1 encoding [3] comes from the observation that after an entropy–reducing transformation (e.g. wavelet, delta, etc.), where the resulting coefficient histogram is centered about zero, most of the redundancy in each coefficient is in the position of the coefficient's leading 1. After the leading 1, the distribution of bits tends to be fairly random. Encoding the position of the leading 1, and passing along unchanged what remains, simplifies the encoding process. Lead–1 encoding also drastically reduces, by several orders of magnitude, the amount of information needed to encode the coefficients. In the case of 12–bit coefficients, instead of 4096 possible symbols, there are 13: one each for each leading–1 position, plus one for the zero coefficient. The encoding table therefore has thirteen 2–byte entries, where each entry give the length of the codeword and the codeword itself. The total encoding table size is 26 bytes—a vast improvement over what would have been required to encode 4096 symbols. The decoding table now in the worst case requires 4096 8–bit entries, or 4 kilobytes.

Lead–1 encoding requires that the data to be encoded first be transformed via an entropy–reducing transformation. Virtually any type of transform may be used, from simple delta–transforms to wavelets. However, for best performance the following two conditions must be met: the transform must produce integer–valued coefficients, and the coefficient histogram must be centered about zero. If the transformation does not naturally produce a histogram centered about zero the coefficients can of course be biased.

4.1 The Encoding Process

The conceptual encoding process is straightforward. Given a coefficient, take its magnitude (absolute value) and determine the coefficient's leading–1 position. A code corresponding to this position is obtained via table–lookup, and placed on an output register. If the magnitude is nonzero, place a sign bit on the output register. Then place the *pass bits*. Pass bits are everything which follows the leading 1. As an example, consider the coefficients 42 and -42. In 16–bit binary, their representation is:

```
 42: 00000000 00101010
-42: 11111111 11010110
```

Take the absolute value of each, and determine the leading 1 position—6, in this case. Obtain the code for a leading–1 position of 6 (say, 0010), and place it on the output register. Place a sign bit on the register indicating the coefficient's sign. Then take the lower 5 (leading–1 position minus 1) bits of the coefficient magnitude and place them on the output register. Thus, for 42 and -42, their encoded output would look like this:

```
 42: 0010 0 01010
-42: 0010 1 01010
```

4.2 The Decoding Process

The decoding process is also straightforward. Given that the longest code length is L, read the first L values of the encoded bitstream, and use them as an index into a table that gives the code length and leading–1 position corresponding to that code. Referencing the example above, after decoding a leading–1 position is 6, initialize a decoded value with a 1 in the correct place:

```
 42: 00000000 00100000
-42: 00000000 00100000
```

The sign bit is read next and noted. Then the pass bits are read and the logical OR is taken with them and the decoded value:

```
 42: 00000000 00101010
-42: 00000000 00101010
```

Finally, if the sign bit indicates a negative value, the result is negated:

```
 42: 00000000 00101010
-42: 11111111 11010110
```

4.3 Determining the Leading–1 Position

The leading–1 position must be determined before it can be encoded. One potential method of determining this is to examine each bit in a coefficient until encountering the first 1. Depending on the coefficient width and the CPU architecture, this may be a slow solution, as it requires a loop with a conditional statement for control. Table lookups are potentially faster. Again, a simple way is to use the 12–bit coefficient as an index and get a table entry. In this case the table would require 4 kilobytes of storage (2^{12} entries at 1 byte each). A more space–efficient way is to create a 256–entry table (8–bit index), requiring 256 bytes total. For each coefficient, if it is < 256 the coefficient is used as an index into the table, and the leading–1 position is returned. Otherwise the coefficient is right–shifted 8 places, the result is used as an index, and 8 is added to the result. Including the 26–byte code table (described in section 4), the total amount of allocated memory required by the code loop is now 282 bytes.

5 Optimizations

Here are some further optimizations to increase overall execution time. These optimizations are for our specific application on our hardware platform. Most of the work developing this coder was performed on a Macintosh PowerMac G5 (IBM PowerPC 970FX v3.1) with a 32-kilobyte L1 data cache. Under these conditions these optimizations have given improved performance. In some cases the improvement is slight but it should be remembered that time is extremely valuable, and so any measurable performance gain is considered worthwhile. An implementation of this encoding method on a different computer architecture or operating system will likely require additional optimizations and adjustments.

5.1 Optimizing Data Access

Instead of taking coefficients directly from the input data stream, buffer them in a 512–byte buffer. This buffering adds a memory copy to the coding loop, but the copy only happens once every 256 coefficients. The speed increase comes from a rather nice side effect of the copy: it pulls the data into cache and aligns it in cache memory. This makes buffer access faster than accessing the input stream directly.

5.2 Streamlining Memory

On MacOS X, the operating system for the PowerMac G5, all memory is allocated in 4 kilobyte pages. Even allocating 1 byte of storage requires a 4 kilobyte page. Instead of allocating memory separately for each table or buffer, an attempted optimization was to allocate a single block of 4 kilobytes, and partition it manually. This requires one page table entry. In practice there was no improvement in program execution with this change.

5.3 Leveraging Hardware to Determine Lead–1 Positions

Many CPU architectures possess a hardware instruction that, given an integer, counts the number of leading zeros in the binary representation of that integer. On the PowerPC architecture this instruction is **cntlzw**. Instead of using a LUT to determine lead–1 positions, leverage the hardware instruction instead, by means of an assembly statement placed inline in the C code. This eliminates the need for the Lead–1 LUT, saving 256 bytes.

5.4 The Zero Short Path

The zero coefficient is a special case. Here there is no sign bit and no pass bits. Due to the entropy–reducing transform that precedes the encoding stage, we expect to see a large number of zero coefficients. This situation presents a great opportunity for further optimizing the coding loop.

Instead of repeatedly looking up the code and code length for the zero coefficient, store them in two variables, initialized at the start of the coding process. During encoding, if the coefficient is detected to be zero, the code is placed directly on the output register, without the need for table lookups. This places a conditional in the program's coding loop, but eliminating table lookups in this very frequent case yields a net speed gain.

5.5 Computation of Sign and Pass Bits When Encoding

Section 4.1 describes the creating of the sign and pass bits. The description given there is conceptual, and while the conceptual description can be implemented directly, the resulting implementation will not be the fastest.

To quickly compute the pass and sign bits at the same time, given both the original coefficient and its magnitude, take the most significant bit (MSB) of the original signed coefficient and shift it down so it is in the coefficient's lead–1 position. Then XOR it with the coefficient magnitude, and place the lower bits onto the output register.

As a more concrete example, again consider the numbers 42 and -42, as in section 4.1. In the following examples, `nPos` refers to the position of the leading 1 in 42 (`nPos` is 6, in this case), and `nMag` refers to the magnitude or absolute value of the coefficients: 42. In binary and hexadecimal representation:

```
 42: 00000000 00101010, 0x002A
-42: 11111111 11010110, 0xFFD6
```

Take the MSB from the signed coefficient and shift it down into the leading–1 position (pseudo C–code notation):

```
 42: (0x002A & 0x8000) >> (16 - nPos) = 0x00
-42: (0xFFD6 & 0x8000) >> (16 - nPos) = 0x20
```

Then XOR it with the coefficient magnitude:

```
 42: 0x00 XOR 0x2A = 0x2A
-42: 0x20 XOR 0x2A = 0x0A
```

The lower `nPos` bits are then placed onto the output register. Note that in this case, on the coded bitstream a sign bit of 1 indicates a *positive* integer, and 0 a *negative* one. This computation of sign and pass bits can be done in one line of C code:

```
((nCoeff & 0x8000) >> (16 - nPos)) ^ nMag;
```

5.6 Reconstructing the Coefficient when Decoding

Given the encoding optimizations above, it is important to note that in the case of a positive coefficient, the sign and pass bits of the encoded coefficient are exactly the same as the coefficient itself. We can use this fact to optimize decoding.

Again referencing the example of encoding and decoding the numbers 42 and -42, assume the leading–1 position has already been decoded, and the sign and pass bits remain on the input buffer:

```
 42: 1 01010
-42: 0 01010
```

Create a binary mask with the value $2^{\text{nPos}} - 1$, used to remove the sign and pass bits off the input buffer. So, for a leading–1 position of 6 the value would be $2^6 - 1 = 0\text{x}3\text{F}$, or 00111111. Mask off the sign and pass bits, and initialize a decoded coefficient to their value:

```
 42: 00101010
-42: 00001010
```

Right–shift the mask by 1, and compare the result to the sign and pass bits:

```
 42: (00101010 <= 00011111)
-42: (00001010 <= 00011111)
```

If the comparison is false, the process is finished—the sign and pass bits are equal to the decoded coefficient, and are placed on the decoded bitstream. If the comparison is true, place a 1 in the leading–1 position and negate the result.

5.7 Encoding Loop Core

What follows is the code for the core of the encoding loop. The code should be understandable, but a few points of explanation are needed. `nOutReg` is a 64-bit register that accumulates output bits as they are generated. `regtype` is a type definition equivalent to an unsigned 64–bit integer. For brevity some code not directly related to encoding has been removed.

```
for (i = numCoeffs; i > 0; i--) {
  nCoeff = *pIn++;
  if (nCoeff != 0) {
    nMag = abs(nCoeff);

    // Get the coefficient's leading
    // 1 position.

    #ifdef ARCH_PPC
      __asm__ __volatile__("cntlzw %0, %1" :
      "=r"(nPos):
       "r"(nMag):"cc");
      nPos = 32-nPos;
    #else
      if (nMag < 256)
          nPos = m_pPosTab[nMag & 0xFF];
      else
          nPos = m_pPosTab[nMag >> 8] + 8;
    #endif

    // Get the 16-bit table entry and
    // extract the needed information.

    nTableEntry = m_pEncTab[nPos];

    nCodeLen = nTableEntry & 0xF;
    nCode = nTableEntry >> 4;

    // Generate the pass bits
    // and the sign bit.

    nPass = ((nCoeff & 0x8000) >>
            (16 - nPos)) ^ nMag;

    // Place code and sign/pass bits
    // on output

    nOutReg |= (regtype)nCode
                << nOutRegCnt;
    nOutRegCnt += nCodeLen;

    nOutReg |= (regtype)(nPass)
                << nOutRegCnt;
    nOutRegCnt += nPos;
     }
    else {

    // This section of code executes
    // when the coefficient is a zero.
    // In that case there's no sign bit
    // nor pass bits.

    nOutReg |= (regtype)nZeroCode
           << nOutRegCnt;
    nOutRegCnt += nZeroCodeLen;
    }
} /* end for(i) */
```

6 Results

All tests were performed on a dual 2.7 GHz PowerMac G5 with 2 gigabytes of RAM, running MacOS X, version 10.4.9. There was only one thread of execution. The test application loads from disk a file containing 100 image frames, where each frame is 4008 x 2672 pixels in size, with each pixel being 12 bits stored in a short integer. The total file size is approximately 2 Gigabytes. The pixels in each frame are transformed via a simple raster–order delta transform, resulting in signed coefficients. Raw data size is therefore 20.42 MB per frame, actual data is 15.32 MB. I quote all compression rates relative to the raw data rate.

6.1 Execution Time

Here is a comparison of execution times for a variety of buffer sizes, while using an inline assembly (hardware) instruction to determine the leading–1 position, compared with using a lookup table to do the same. Timings do not include disk access, and only measure the encoding process—the transform performed prior to encoding is not timed.

The results are given in table 1. The first thing noticable is that if a processor instruction is used for determining the leading–1 position, it is better not to explicitly buffer the input. I do not know at this time why this is the case. The result is that when the hardware instruction is used, only 26 bytes of allocated memory are required for the encoder. The second thing to note is that when buffering input a small buffer (256 entries, or 512 bytes) is more efficient than an even slightly larger one. Finally, as expected, using a hardware instruction to determine the lead–1 position is always faster than using a lookup table.

	Table			cntlzw		
Buffer Entries	Time (s)	Throughput	Memory	Time (s)	Throughput	Memory
None	14.59	140.0	282	12.50	163.4	26
256	14.25	143.3	794	12.80	159.6	538
512	14.76	138.4	1306	13.08	156.2	1050
768	15.03	135.9	1818	13.20	154.74	1562
1024	14.75	138.5	2330	13.20	154.74	2074
2048	14.75	138.5	4378	13.25	154.16	4122
4096	14.72	138.8	8474	13.29	153.7	8218

Table 1: Execution times for various buffer sizes, comparing the use of tables vs the hardware instruction **cntlzw** to determine the leading–1 position. Throughput is measured in megabytes per second, and is based on the raw data rate. Total allocated memory is given in bytes.

6.2 Comparison to Other Coders

The 100–frame sequence was also compressed using the standard compressors bzip2 and gzip, and the well–known range coder by Michael Schindler [7]. These were compared to the coder described in this report (referred to as "Lead–1"). All were timed using the command–line utility **time**. Results are given in table 2. These results and those in table 1 show that for the purpose of quickly and efficiently compressing this sort of raw image data it is neither necessary nor desirable to use a sophisticated compression technique. A well–thought–out, simple method is both efficient and extremely fast.

7 Conclusions and Future Work

This report presented the reasoning behind and implementation of a high–speed entropy coder, suitable for rapid lossless compression of EO sensor data. Leveraging existing hardware capabilities, where possible, can have a dramatic positive effect on the the system. In this case, making a call to a single hardware instruction increased throughput by up to 23 megabytes per second and reduced memory requirements 26 bytes, making the coder's data cache footprint truly compact. Even without the hardware instruction, allocated memory was only 794 bytes, which is still tiny. The coder has not yet been added to the software processing pipeline, but it or some other coder will likely be added in the future.

The weakness of the coder presented here is its use of Canonical codes. Canonical codes cannot code a symbol to less than 1 bit. This means that the coder is incapable of encoding a 20 MB image frame to anything less than 1.28 MB, or 6.25% of original size.

It is desirable to have a coder capable of less than 1 bit per symbol. There are two promising alternatives: coders utilizing State–Tree Codes [9], and Length–Limited Variable–to–Variable Length Codes [8]. These coders are simple, faster than arithmetic–type coders, and achieve a compression rate that is comparable to or better than arithmetic–type coders. Future efforts will be focused on how to engineer the coders to achieve the high throughput which we require.

8 Acknowledgements

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and partially by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] Martin Burtscher and Paruj Ratanaworabhan. High throughput compression of double–precision floating–point data. In *Proceedings 2007 Data Compression Conference (DCC'07)*, pages 293–302. IEEE Computer Society, 2007.

[2] Jr Glen G. Langdon. Adaptive binary arithmetic coding for multi–media applications. In *COMPCON Spring '91 Digest of Papers*, pages 354–357, 1991.

[3] B. Gregorski, J. Senecal, M.A. Duchaineau, and K.I .Joy. Adaptive extraction of time–varying isosurfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):683–694, 2004.

[4] Paul G. Howard and Jeffrey S. Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6):857–865, Jun 1994.

[5] G.N.N. Martin. Range encoding: an algorithm for removing redundancy from a digitised mes-

	Raw Pixels		Delta Transformed	
Coder	Size	Time	Size	Time
bzip2	1175.60	654.105	1150.93	771.117
gzip	1577.37	473.657	1426.78	311.418
Range	1677.54	96.956	1414.36	94.980
Lead–1	—	—	1128.95	19.184

Table 2: Comparison of execution times and compressed size for some common encoders. All times are in seconds, and are user time as reported by the utility **time**. Compressed sizes are in Megabytes.

sage. In *International Conference on Video and Data Recording*, pages 187–197, 1979.

[6] Alistair Moffat and Jyrki Katajainen. In–place calculation of minimum-redundancy codes. In *Algorithms and Data Structures. 4th International Workshop, WADS '95*, pages 393–402. Springer-Verlag, 1995.

[7] Michael Schindler. A fast renormalisation for arithmetic coding. In *Proceedings DCC '98 Data Compression Conference*, page 572. IEEE Computer Society, IEEE Computer Society Press, 1998.

[8] Joshua Senecal, Mark A. Duchaineau, and Kenneth I. Joy. Length–limited variable–to–variable length codes for high–performance entropy coding. In James A. Storer and Martin Cohn, editors, *Proceedings, Data Compression Conference (DCC 2004)*, pages 389–398. IEEE Computer Society, Mar 2004.

[9] Peter R. Stubley. Adaptive variable-to-variable length codes. In *DCC '94: Data Compression Conference*, pages 98–105, 1994.

[10] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, Jun 1987.