

A Second Modified Run Length Encoding Scheme for Blocksor Transform ed Data

by Michael A. Maniscalco

This paper details a run length encoding (RLE) scheme which is designed to work with data streams which have undergone blocksorting. This RLE method is an extension of the simpler RLE method which I had described previously in the paper [A Modified Run Length Encoding Scheme For Blocksor Transform ed Data](#).

This paper does not detail the basics of Run Length Encoding. For those who are not familiar with Run Length Encoding, there are many papers available at Mark Nelson's "Data Compression Library" located at <http://www.dogma.net/DataCompression>.

The Modified Run Length Encoding Strategy

The basic concept behind the Modified Run Length Encoding method (mod-RLE) is to modify the lengths of the symbol runs in the source data stream such that the resulting run length can be used to determine the length of the original run. Additional information is required to reverse the encoded run and, this information is stored elsewhere in the encoded stream. For the original mod-RLE (mod1-RLE), this additional information was a fixed length code of 1 bit in size. For the second mod-RLE (mod2-RLE), this additional information is a variable length code.

History: The Original MOD1-RLE

The original Mod1-RLE method employed fixed length codes of 1 bit in size to reverse the actual Run Length Encoding. The transform was employed only on runs of length 3 or greater where the encoded run is of length $2 + ((N-1)/2)$ and N is the length of the original run. To reverse this, the additional fixed length code of 1 bit is needed and is either 0 if the original run was odd in length or, 1 if the original run was even in length. Thus, the decoder could reverse the mod-RLE with the formula $((N-2)*2)+1$ where N is the length of the encoded run. And this value is incremented by 1 if the fixed length 1 bit code has a value of 1.

Thus, in C the code might appear as:

```
// for encoding ...
tempRunLength = originalRunLength + 3;
fixedLengthCodeBit = tempRunLength & 1;
encodedRunLength = tempRunLength >> 1;

// for decoding ...
originalRunLength = ((encodedRunLength << 1) |
fixedLengthCodeBit)-3;
```

Compression ratios can be improved here by using an Arithmetic encoder to encode the single bit fixed length code because, after blocksorting, data streams will usually contain far more runs of length 3 than length 4. Thus, this bit is more often 0 than 1. But, for the sake of simplicity, the original paper on Mod-RLE1 did not use this improvement.

Improvements: The MOD2-RLE

The remainder of the paper will focus on the new Mod2-RLE. This method has several improvements over the original Mod1-RLE method. These improvements include the introduction of variable length codes rather than the fixed length 1 bit codes used in Mod1-RLE as well as the employment of arithmetic encoding for these variable length codes. The resulting encoded streams produced with Mod2-RLE are typically much smaller than those encoded with Mod1-RLE. However, while the Mod2-RLE streams are more compressed than the Mod1-RLE streams, the overall compression results where RLE is only the first step in the compression process, (i.e. blocksorting) are typically the same as the original Mod1-RLE. This came as quite a surprise but these results might be due to the post-RLE compression scheme used in this test*. There are some cases where Mod1-RLE is slightly better than the more complex Mod2-RLE. But, in general, Mod2-RLE does typically lead to moderate compression improvements.

* **NOTE:** The post-RLE compression scheme used here is M99. This coder is noted to typically produce more compressed streams than a general purpose RLE scheme. I believe this is why the overall compression rates for Mod1-RLE and Mod2-RLE are so similar. Further tests to prove this will involve using a standard Arithmetic coder in stead of an M99 coder.

The general strategy with Mod2-RLE is somewhat similar to its predecessor. As with Mod1-RLE, only runs of length 3 or greater will be encoded. Also, as with Mod1-RLE, the length of the resulting encoded run will be used to determine the length of the original run. However, with Mod2-RLE the length of the encoded run will also be used to determine the size of the variable length code (stored else where in the compressed stream) which is needed to realize the original decoded run length.

The following table illustrates how the size and value of the variable length codes for Mod2-RLE are derived.

Original Run Length (ORL)	ORL - 1	ORL - 1 (Binary)	Bit Length	Bit Length -1 Variable Length Code Size	From ORL -1 (Binary) Variable Length Code Value
3	2	<u>10</u>	2	1	<u>0</u> = 0
4	3	<u>11</u>	2	1	<u>1</u> = 1

5	4	<u>100</u>	3	2	<u>00</u> = 0
6	5	<u>101</u>	3	2	<u>01</u> = 1
7	6	<u>110</u>	3	2	<u>10</u> = 2
8	7	<u>111</u>	3	2	<u>11</u> = 3
9	8	<u>1000</u>	4	3	<u>000</u> = 0
10	9	<u>1001</u>	4	3	<u>001</u> = 1
11	10	<u>1010</u>	4	3	<u>010</u> = 2
12	11	<u>1011</u>	4	3	<u>011</u> = 3
13	12	<u>1100</u>	4	3	<u>100</u> = 4
14	13	<u>1101</u>	4	3	<u>101</u> = 5
15	14	<u>1110</u>	4	3	<u>110</u> = 6
16	15	<u>1111</u>	4	3	<u>111</u> = 7
17	16	<u>10000</u>	5	4	<u>0000</u> = 0
18	17	<u>10001</u>	5	4	<u>0001</u> = 1

Thus, in C, the code for calculating the size of the variable length code might be:

```

if (originalRunLength >= 3){
    orlMinusOne = originalRunLength - 1;
    codeSize = -1;
    bit = 1;
    while (bit <= orlMinusOne){
        codeSize++;
        bit <<= 1;
    }
}
else
    codeSize = 0;

```

And the code for calculating the value of the variable length code might be:

```

codeValue = 0;
bit = 1;
while (codeSize > 0){

```

```

codeValue |= ( orlMinusOne & bit);
bit <<= 1;
codeSize--;
}

```

Of course, it would be wiser to build a look up table to improve speed rather than to re-calculate these values with every encoding.

At this point, all that remains in order to encode a run is to calculate the new run length for the encoded stream. This new run length is calculated as $N+2$ where N is the number of bits in the variable length code as calculated above. Thus, an original run length of 3 or 4 generates an encoded run length of 3. An original run length of 5,6,7 or 8 generates an encoded run length of 4. An original run length of 9 through 16 generates a run length of 5. Etc.

The decoding process determines the length of the original run as follows:

The size of the variable length code is $N-2$ bits where N is the length of the encoded run.

The length of the original run is then calculated as $2^{(N-2)}+V+1$ where N is the length of the encoded run and V is the value of the variable length code.

The following table illustrates the decoding process.

Encode Run Length (N)	Variable Length Code Size (S) = N-2	Variable Length Code Value (V)	Decoded Run Length $(2^S)+V+1$
3	1	0 = 0	$(2^1)+0+1 = 3$
3	1	1 = 1	$(2^1)+1+1 = 4$
4	2	00 = 0	$(2^2)+0+1 = 5$
4	2	01 = 1	$(2^2)+1+1 = 6$
4	2	10 = 2	$(2^2)+2+1 = 7$
4	2	11 = 3	$(2^2)+3+1 = 8$
5	3	000 = 0	$(2^3)+0+1 = 9$
5	3	001 = 1	$(2^3)+1+1 = 10$
5	3	010 = 2	$(2^3)+2+1 = 11$

5	3	011 = 3	$(2^3)+3+1 = 12$
5	3	100 = 4	$(2^3)+4+1 = 13$
5	3	101 = 5	$(2^3)+5+1 = 14$
5	3	110 = 6	$(2^3)+6+1 = 15$
5	3	111 = 7	$(2^3)+7+1 = 16$
6	4	0000 = 0	$(2^4)+0+1 = 17$
6	4	0000 = 1	$(2^4)+1+1 = 18$

Thus, the code for the decoder might be:

```
int calc2Exp(int N){
    // calculate 2^N
    int v = 1;
    while (N>0){
        N--;
        v <<= 1;
    }
    return v;
}

if (encodedRunLength >= 3){
    variableLengthCodeSize = encodedRunLength - 2;
    variableLengthCodeValue = getCode(variableLengthCodeSize);
    decodedRunLength = calc2Exp(variableLengthCodeSize) +
    variableLengthCodeValue + 1;
}
else
    decodedRunLength = encodedRunLength;
```

Adding Arithmetic Coding:

As mentioned above, a strong improvement to Mod2-RLE would be to encode the variable length code values using an arithmetic encoder. Since most of the variable length codes will be less than 8 bits in size (often far less), the strategy that I implemented was to have two variable length code buffers. The first is the output stream from an arithmetic coder which is used to encode all variable length codes of size 8 bits and less. The second buffer is used to store the literal binary values of codes which are greater than 8 bits in length.

The following table is the results of both the original Mod1-RLE and the new Mod2-RLE presented here on the Calgary corpus as apply to the files post-blocksort. Also

included is the compression ratio when the M99 encoding scheme is applied to the Mod-RLE encoded streams.

File	Size	Mod1-RLE	Mod2-RLE	Mod1-RLE+M99	Mod2-RLE+M99
bib	111,261	83,090	64,737	28,610	28,329
book1	768,771	650,338	592,170	225,833	226,963
book2	610,856	479,106	403,311	157,242	156,932
geo	102,400	88,276	76,550	58,247	58,196
news	377,109	304,769	264,416	121,117	120,559
obj1	21,504	17,638	15,006	10,867	10,913
obj2	246,814	183,239	142,395	80,699	79,878
paper1	53,161	43,265	38,654	17,305	17,342
paper2	82,199	67,517	60,254	25,561	25,545
paper3	46,526	39,429	36,463	16,240	16,328
paper4	13,286	11,660	11,042	5,343	5,431
paper5	11,954	10,386	9,836	5,019	5,098
paper6	38,105	31,354	28,408	12,912	13,007
pick	513,216	301,372	115,138	46,752	47,771
prig	39,611	31,907	28,265	13,122	13,150
prowl	71,646	52,265	41,197	16,720	16,648
prop	49,379	35,921	28,244	11,903	11,794
trans	93,695	64,589	47,007	20,548	20,282
Totals	3,251,493	2,496,121	2,003,093	874,040	874,166

Conclusion:

The results of the two Mod-RLE are very similar. It is also noteworthy that the Mod1-RLE method does not call for the use of Arithmetic coding where the Mod2-RLE method does. Tests suggest that the Mod1-RLE method would be improved by approx 2,000 bytes by employing an arithmetic coder. While this would make the original Mod1-RLE method a better method overall, this does not suggest that it is a better choice in all cases. Note that the overall size after just Mod-RLE is better for Mod2-RLE. This suggests that it is a better choice for pre-blocksort RLE where Mod1-RLE is a better choice for post-blocksort RLE.

(C) 2001 Michael A Maniscalco