Burrows-Wheeler Compression with Variable Length Integer Codes

Peter Fenwick *

July 10, 2002

Abstract

The final coder in Burrows-Wheeler compression is usually either an adaptive Huffman coder (for speed) or a complex of arithmetic coders for better compression. This article describes the use of conventional pre-defined variable length codes or universal codes and shows that they too can give excellent compression. The paper also describes a "sticky Move-to-Front" modification which gives a useful improvement in compression for most files.

Keywords
Burrows-Wheeler compression, Elias gamma codes, Fibonacci codes, Fraenkel-Klein codes, universal codes.

1 Introduction

In less than a decade the Burrows-Wheeler algorithm has become a standard for fast efficient lossless data compression. While the algorithm has been the subject of many papers, there has been little more than 7% improvement in compression since its first publication.

Many of the improvements require complex encoding structures and extensive analysis of the final stream of encoded values. This paper goes to the

*Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand. Email : p.fenwick@auckland.ac.nz

other extreme, showing that a very simple final coder based on established variable length codes for the integers can still give excellent compression.

The paper also describes an improvement to the "Move-to-Front" step within Burrows-Wheeler compression, an improvement which has been informally communicated and used in production compressors but not otherwise presented in the open literature.

2 Burrows-Wheeler compression

The compression algorithm described by Burrows and Wheeler[1] differs from most compression methods in that it processes a file in blocks. While a block size of a megabyte is quite practical on most computers, it is often convenient to work with blocks of say 200 kilobytes, so that a 1 megabyte file would be processed as 5 blocks. Within each block, compression proceeds in three major phases.

1. An initial permutation step (the "Burrows-Wheeler Transform") reorders the input symbols of each block (or file) according to their adjoining contexts. Because most contexts are associated with only a few symbols, the transformed block contains long sequences with few symbols and has many runs of repeated symbols.

2. The essence of the sequences of few symbols is captured by a "recency" recoding in which each symbol is replaced by the number of *different* symbols encountered since its last occurrence. The usual implementation of a recency recoder uses a Move-to-Front list, with each symbol being recoded as its index in the list. The symbol is then brought to the head of the list, with all intervening symbols shuffled back one place. The output of this step is a series of integers, dominated by small values. Typically about 60% are 0 (from symbol runs) and for many files a value n has a frequency almost proportional to $1/n^2$[6].

3. The highly skewed distribution from step 2 is captured by a final statistical coder, often a form of Huffman or arithmetic coder, to ensure that the more frequent small values have an appropriately compact representation.

There are thus three essential steps to the compression. All have been investigated by various workers, with a view to improving either speed or the

Author	date	algorithm	bpc
Burrows & Wheeler	1994	16 k Huffman blocks	2.43
Fenwick	1995	order-0 arithmetic	2.52
Fenwick	1996	structured arithmetic	2.34
Balkenhol et al	1998	cascaded arithmetic	2.30
Fenwick	1998	sticky MTF, struct arith	2.30
Balkenhol & Shtarkov *	1999	cascaded arithmetic	2.26
Deorowicz	2000	arith. contexts	2.27
Deorowicz *	2002	weighted frequency count	2.25
Seward (BZIP2)	2000	Huffman blocks	2.37
Wirth	2001	no MTF; PPM contexts	2.35
Arnavut *	2002	inversion ranks	2.30
Fenwick (this paper)	2002	VL codes; 1 k blocks	2.57

Table 1: Burrows-Wheeler compression history

compression performance. Some of the earliest work was done by the author with a series of reports [2, 3, 4] and two papers [5, 6]. Major surveys of Burrows-Wheeler compression are given by Balkenhol, Kurtz and Shtarkov[7] and Deorowicz[8], to which the reader is referred.

Most Burrows-Wheeler compressors use some form of adaptive statistical encoder for the final stage. Burrows and Wheeler, and more recently Seward[13] with BZIP2, use Huffman codes, recomputed for successive blocks of a few thousand symbols. Most other workers have used complexes of arithmetic coders in an attempt to get the best possible compression, albeit with a considerable increase in complexity. These techniques are described by Fenwick[5, 6], Balkenholl et al[7] and Deorowicz[8].

The compression performance (in bpc, or "bits per character") for the Calgary Corpus is summarised in Table 1. It is obvious that improvements in compression are difficult, with even the best result improving on the early 1996 results by only 4%.

The last four lines show some other important results, aside from the historical sequence of the preceding lines. The table includes results from Wirth and Moffat[12] (who avoid the MTF stage completely), Seward's BZIP2 implementation (released through the Free Software Foundation) and the results of this paper. Three recent results, marked with a * in the table, indicate the state of the art at mid-2002.

Arnavut [9] uses "inversion ranks" as an alternative to Move-To-Front re-

coding. This, the most recent of a series of papers, shows that inversion ranks are clearly competitive with Move-To-Front.

Balkenhol & Shtarkov [10] demonstrate an improvement over their earlier results[7].

Deorowicz [11] examines the Move-To-Front operation, replacing it by a "Weighted Frequency Count" to obtain the best result to date.

2.1 Present Work

This current paper arose from a major survey of variable-length codes for the integers in which many of the codes were compared as the final coding step of a Burrows-Wheeler compressor. The results were sufficiently encouraging to justify reporting in this paper.

Before discussing the compression proper it is appropriate to briefly describe the two variable-length codes which are used in the coder. We also describe a simple modification of the Move-to-Front coder which gives a useful improvement in compression.

3 Variable length codes

Variable-length codes (or universal codes) are a special class of integer representations which represent values as compactly as possible, while including some means of terminating each representation or codeword.

The codes used here exemplify two approaches to variable-length codes.

- The Elias γ codes use explicit control bits, either embedded within the main codeword or as a prefix, to indicate the codeword length.

- The Fraenkel-Klein codes, based on Fibonacci numbers, constrain the legal bit pattern within the representation and violate this constraint to terminate the codeword.

3.1 Elias Gamma Codes

These are one of the older variable-length representations, dating from a 1975 paper by Elias[14]. The two versions of the Elias γ code have identical performance, differing only in the permutation of the bits.

Value	γ	γ'	Fibonacci
1	1	1	11
3	011	011	0011
5	01001	00101	00011
7	01011	00111	01011
11	0101001	0001011	001011
13	0100011	0001101	0000011
17	010000001	000010001	1010011
23	010101001	000010111	01000011
127	0101010101011	0000001111111	10100001011
256	00000000000000001	00000000100000000	0100001000011

Table 2: Examples of Elias γ and Fibonacci codes

1. The γ code proper consists of the bits of the binary representation, least-significant bit first, and ending with the most-significant 1-bit. Each numeric bit, except the final 1, is preceded by a 0 "flag" bit. Thus $7 \rightarrow 01011$ and $22 \rightarrow 000101001$.

   To decode, read the flag bits and assume that each '0' flag is followed by a numeric bit; the final '1' flag implies the most-significant 1 on the binary representation, as well as marking the end.

2. Alternatively, the γ' code writes the numeric bits in order, most-significant first. These bits are preceded by as many zeros as there are bits following that most-significant 1. With this code $7 \rightarrow 00111$ and $22 \rightarrow 000010110$.

3.2 Fraenkel-Klein Fibonacci codes

The Fibonacci numbers are a well known number sequence in which each number is the sum of its two predecessors, the first few being 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

$$F_n = F_{n-1} + F_{n-2}, \quad F_2 = F_1 = 1$$

Zeckendorf's theorem [15] states that any integer can be represented as a unique sum of Fibonacci numbers; we write the representation as a binary

bit vector with a 1 indicating the presence of that Fibonacci number (least-significant bit first and omitting F_1). Thus $7 = 2 + 5 = F_3 + F_5 \rightarrow 0101$ and $17 = 1 + 3 + 13 = F_2 + F_4 + F_7 \rightarrow 0101001$.

No Zeckendorf representation has two consecutive ones. By the definition of Fibonacci numbers, any such a pair could be replaced by a single more-significant 1. Thus if we present the digits in increasing significance and follow the most-significant 1 by another 1, this "11" pair is an illegal combination and can act as a terminator to the codeword. This gives Fraenkel and Klein's "C1" code[16], which we will refer to simply as a "Fibonacci" code.

A few examples of the γ and Fibonacci codes are shown in Table 2, mostly for the first few prime numbers. The γ codewords are shorter for small values, but the Fibonacci codewords are shorter for larger values. More precisely, for values of n binary digits, the γ code requires $(2n - 1)$ bits, whereas the Fibonacci code requires about $(1.44n + 0.5)$ bits.

3.3 Wheeler 1/2 code

It is usual to run-length encode the long sequences of zeros produced by the MTF recoding. This coding is not absolutely necessary for fully-adaptive arithmetic encoders, although it does ease their need for fast adjustment, but is essential for Huffman and similar coders which cannot represent an input bit by less than one output bit.

For this application Wheeler has used an unusual code which he calls the "1/2" code and states that while he has used it for a long time he does not know its origin. (The author has had frequent requests asking how it works!)

The code is designed to represent a value as a sequence of zeros and ones (values, not bits) within a stream of larger values. Any value other than 0 or 1 terminates the encoded value. In a normal binary representation all zeros carry a numeric weight of 0, while ones have successive weights of 1, 2, 4, 8, 16, ... In the Wheeler 1/2 code successive zeros have the weights 1, 2, 4, 8, 16, ..., while successive ones have the weights 2, 4, 8, 16, 32, ... While *decoding* a Wheeler 1/2 representation is quite simple, it is much less obvious how the code is generated in first place; for example should the value 2 be represented (least-significant bit first) by 1 or by 01?

To explain the code the bit weights can be written as –

	bit=0	1	2	4	8	16	32	64	...
	bit=1	2	4	8	16	32	64	128	...
OR	bit=1	1+1	2+2	4+4	8+8	16+16	32+32	64+64	...

run	length L	sums	code	$L+1$ binary
x 0 y	1	1	0	01
x 00 y	2	2	1	11
x 000 y	3	1+2	00	001
x 0000 y	4	2+2	10	101
x 00000 y	5	1+4	01	011
x 000000 y	6	2+4	11	111
x 0000000 y	7	1+2+4	000	0001

Table 3: The Wheeler 1/2 code

Thus bit position i has a constant weight of 2^i, to which is added a further 2^i if the digit is 1. For a k-digit Wheeler 1/2 value the constant weights add to $2^k - 1$; if we encode not L but $(L+1)$ the $2^k - 1$ becomes 2^k, which is simply a further bit beyond those of the Wheeler representation. To encode an integer length L into the Wheeler 1/2 code, just encode $(L+1)$ in binary and discard the most-significant 1 bit.

This interpretation is seen immediately in Table 3. Interestingly, the author finds it simpler to *encode* from binary, but to *decode* with the "peculiar" weights.

The Wheeler 1/2 code *never* expands the run which it represents, although all values $N > 1$ must be encoded as $N + 1$ to accommodate the two values used for zeros. As most of the variable-length codes represent a minimum value of 1, we must use the values 1 and 2 rather than 0 and 1 for the Wheeler 1/2 run-length representation, and encode all values $N > 0$ as $N + 2$.

4 Sticky Move-to-Front

Many authors see a major problem in the asymmetry of the Move-to-Front transformation; it brings a "new" symbol into consideration very quickly, but forgets it quite slowly. A less-active symbol is displaced only because it drifts slowly out of consideration as more-recent symbols come ahead of it in the MTF list. The initial fast motion towards the front contrasts with the later slow movement away from the front.

Close observation of the MTF behaviour shows that while most symbols

remain active for a while after they are introduced, many are used once and then not again for a long time. But having been brought to the head of the list, and while they are drifting back to less-active status, they stay ahead of genuinely-active symbols and increase their coding cost.

The "sticky Move-to-Front" implementation reduces this effect by moving the most recent symbol away from the list head if it has been used only once, but keeping it reasonably close to the head just in case it is needed again. It is left at the head of MTF list if it has been referenced twice in succession, producing a 0 MTF recoding on the second reference.

Experiments show that moving a symbol back to 40% of its original position is a good compromise and gives an average improvement of about 1% on both the Calgary and Canterbury compression corpora[1].

5 Compression Measurements

A standard Burrows-Wheeler compressor was modified to emit integers as variable-length codes instead of the more efficient arithmetic codes. Initially a single code (say Elias γ or Fibonacci) was used to encode all the values, producing the "γ" and "Fibonacci" results of Table 4. An immediate observation is that generally the text files (with an alphabet less than 100 symbols) compress better with a γ code, while other files are better with a Fibonacci code.

The next version selected the code (γ or Fibonacci) according to the data statistics of blocks of values. (These *coding* blocks are quite different from the *permutation* blocks of the Burrows-Wheeler transform itself.) Many earlier workers have emitted code in blocks, with code parameters selected for each block. For example, Burrows & Wheeler (the original paper) and Seward (BZIP2) emit in blocks of a few thousand symbols, generating an appropriate Huffman code for each block. The combination of blocks and arithmetic coders is generally less successful because the arithmetic coders adapt quickly to the statistics of each block and optimising according to the block statistics affects only the block "start-up" behaviour which is a relatively small component of the total coding cost.

Here we examine the next say 1000 *non-zero* symbols and calculate the costs of encoding that block in each of the candidate codes. The best code is selected and a control value emitted if necessary to switch to the new code.

[1]By such increments is compression improvement measured!

File		Fibo-	Elias	simple	Vrbl Len blocks		BZIP2
bytes	name	nacci	γ		sticky	n-stick	
111 261	bib	2.202	2.178	2.178	2.150	2.168	1.975
768 771	book1	2.939	2.857	2.857	2.806	2.843	2.420
610 856	book2	2.425	2.344	2.344	2.309	2.332	2.062
102 400	geo	5.267	6.090	5.267	5.267	5.380	4.447
377 109	news	2.770	2.779	2.779	2.712	2.744	2.516
21 504	obj1	4.047	4.528	4.047	4.047	4.124	4.013
246 814	obj2	2.587	2.741	2.587	2.573	2.593	2.478
53 161	paper1	2.751	2.719	2.719	2.675	2.687	2.492
82 199	paper2	2.786	2.739	2.739	2.697	2.722	2.437
513 216	pic	0.832	0.867	0.832	0.826	0.850	0.776
39 611	progc	2.733	2.716	2.716	2.679	2.692	2.533
71 646	progl	1.932	1.846	1.846	1.835	1.847	1.740
49 379	progp	1.903	1.835	1.835	1.829	1.823	1.735
93 695	trans	1.644	1.581	1.581	1.578	1.579	1.528
	Average	2.630	2.701	2.595	2.570	2.599	2.368

Table 4: Compression results, Calgary Corpus

This calculation must be done precisely, including the cost of specifying the new code and especially the cost of the encoded runs of zeros. Several variable length codes were considered during these comparisons, but only the Elias γ and Fibonacci were ever selected. A block size of 1000–2000 gives good performance. The results in Table 4 use a block of whatever size contains 1000 non-zero values.

- Most text files use Fibonacci codes at the start. The early parts of the permuted text files are associated with punctuation and "special character" contexts which are very poor in defining associated characters. The MTF coded values tend to be larger and more variable, so preferring the Fibonacci code which has shorter codewords for large values.

- The later parts of most text files generally use γ codes because their alphabetic contexts are much better at predicting symbols and produce a predominance of small values. The γ codes are better in this situation.

- The binary files, with a weaker structure of contexts and a much wider

File		Fibo-	Elias	simple	Vrbl Len blocks		BZIP2
bytes	name	nacci	γ		sticky	n-stick	
11 150	csrc	2.281	2.244	2.244	2.241	2.228	2.180
1 029 744	excl	1.996	2.246	1.996	1.919	2.060	1.074
513 216	fax	0.832	0.867	0.832	0.826	0.850	0.781
24 603	html	2.619	2.676	2.619	2.618	2.643	2.479
3 721	lisp	2.723	2.754	2.723	2.723	2.714	2.758
4 227	man	3.326	3.380	3.326	3.326	3.334	3.335
125 179	play	2.986	2.947	2.947	2.889	2.920	2.529
481 861	poem	2.938	2.881	2.881	2.824	2.858	2.416
38 240	sprc	2.771	2.896	2.771	2.764	2.795	2.717
426 754	tech	2.389	2.303	2.303	2.265	2.290	2.018
152 089	text	2.671	2.589	2.589	2.560	2.584	2.275
Average		2.503	2.526	2.476	2.450	2.480	2.233

Table 5: Compression results, Canterbury Corpus

range of values, are handled almost entirely by Fibonacci codes and show little improvement over the simpler scheme of selecting the Fibonacci code for non-text files.

The column "n-stick" shows the performance with sticky-MTF disabled. Most files show an improvement with sticky-MTF of 0.02–0.03 bpc, with an overall improvement of just over 1%, although a few show slightly worse compression.

Table 5 gives corresponding results for the newer Canterbury Corpus. The compression is 9.7% poorer than with BZIP2 (compare 8.4% for the Calgary corpus). Sticky-MTF improves compression by 1.2% (Calgary 1.1%).

But the EXCL file is quite anomalous. With most files the new compressors are within 10% of the BZIP2 results, but EXCL gives about *half* the compression of BZIP2 (2 bpc cf 1 bpc). Figure 1 shows the distribution of encoded values (after run-length encoding the zeros) for three files. While the files POEM and GEO show the typical, nearly monotonic, decrease in symbol frequency, the EXCL file is completely different. Its frequency distribution in no way approximates that assumed by the use of non-adaptive variable-length codes, but has significant peaks and is actually increasing for the largest values! Many of the more-frequent large values must be encoded with relatively costly variable-length representations, whereas an adaptive

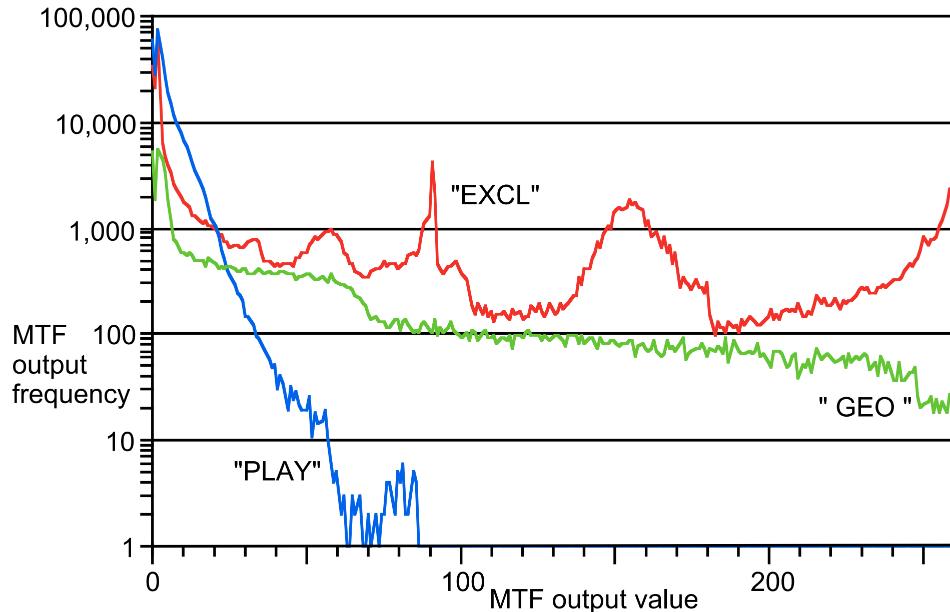

Figure 1: M T F code frequencies.

Huffman or arithmetic coder could compensate for these unexpected frequencies. The compression is correspondingly poorer for the non-adaptive variable-length codes.

In results not shown, enabling sticky MTF with a good final compressor *degrades* the compression of EXCL by 22%! This phenomenon has not been investigated, but probably arises from the unexpected frequency distribution.

6 Conclusions

We have shown that simple variable-length codes are a viable alternative to the more usual arithmetic or block-adaptive Huffman coders usually employed in a Burrows-Wheeler compressor, giving compression within 10% of the production BZIP2 compressor and 14% of the best reported results. We have also described a version of the Move-to-Front recoder which gives a 1% improvement on each of the two standard corpora.

7 Acknowledgments

The author thanks the referees for their constructive comments and for suggesting that some recent results be added to Table 1.

References

[1] Burrows M., Wheeler D.J. (1994) "A Block-sorting Lossless Data Compression Algorithm", *SRC Research Report 124*, Digital Systems Research Center, Palo Alto. gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z

[2] Fenwick, P.M., "Experiments with a Block Sorting Text Compression Algorithm", *The University of Auckland, Department of Computer Science*, Report No 111, May 1995.

[3] Fenwick, P.M., "Improvements to the Block Sorting Text Compression Algorithm", *The University of Auckland, Department of Computer Science*, Report No 120, Aug. 1995.

[4] Fenwick, P.M., "Block Sorting Text Compression – Final Report", *The University of Auckland, Department of Computer Science*, Report No 130, Apr. 1996.

[5] Fenwick, P.M. (1996) "Block Sorting Text Compression", *ACSC-96 Proc. 19th Australasian Computer Science Conference*, Melbourne Jan 1996. pp 193–202

[6] Fenwick, P.M., "The Burrows-Wheeler Transform for Block Sorting Text Compression – Principles and Improvements", *The Computer Journal*, Vol 39, No 9, pp 731–740, 1996.

[7] Balkenhol, B., Kurtz,S., and Shtarkov, Y.M., "Modifications of the Burrows and Wheeler data compression algorithm", *Proc. IEEE Data Compression Conference*, March 1999, IEEE Computer Society Press, Los Alamitos, California, pp188–197.

[8] Deorowicz, S., "Improvements to Burrows-Wheeler Compression Algorithm", *Software – Practice and Experience*, Vol 30, No 13, Nov 2000, pp 1465–1483.

[9] Arnavut, A, "Generalization of the BWT Transformation and Inversion Ranks", *Proc. IEEE Data Compression Conference*, March 2002, IEEE Computer Society Press, Los Alamitos, California, pp477–486.

[10] Balkenhol, B. and Shtarkov, Y.M., "One attempt of a compression algorithm using BWT", (Unpublished report) `www.mathematik.uni-bielefeld.de/sfb343/preprints/pr99133.ps.gz`

[11] Deorowicz, S., "Second step algorithms in the Burrows-Wheeler compression algorithm", *Software – Practice and Experience*, Vol 32, No 2, 2002, pp 99–111.

[12] Wirth, A.I., and Moffat, A., "Can we do without ranks in Burrows-Wheeler Transform compression?", *Proc. IEEE Data Compression Conference*, March 2001, IEEE Computer Society Press, Los Alamitos, California, pp 419–428.

[13] Seward, J. (1996) Private Communication. For notes on the released BZIP see http://hpux.cae.wisc.edu/man/Sysadmin/bzip-0.21

[14] Elias, P., "Universal Codeword Sets and Representations of the Integers", *IEEE Trans. Info. Theory*, Vol IT 21, No 2, pp 194–203, Mar 1975

[15] E. Zeckendorf, "Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas", *Bull. Soc. Roy. Sci. Liège*, Vol 41 (1972), pp 179–182

[16] A.S. Fraenkel and S.T. Klein, "Robust universal complete codes for transmission and compression", *Discrete Applied Mathematics* Vol 64 (1996) pp 31–55.