The Burrows–Wheeler Transform for Block Sorting Text Compression—Principles and Improvements

Abstract

A recent development in text compression is a "block sorting" algorithm which permutes the input text according to a special sort procedure and then processes the permuted text with Move-to-Front and a final statistical compressor. The technique combines good speed with excellent compression performance.

This paper investigates the fundamental operation of the algorithm and presents some improvements based on that analysis.

Although block sorting is clearly related to previous compression techniques, it appears that it is best described by techniques derived from work by Shannon in 1951 on the prediction and entropy of English text. A simple model is developed which relates the compression to the proportion of zeros after the MTF stage.

Short Title Block Sorting Text Compression

Author Peter M. Fenwick

Affiliation
Department of Computer Science
The University of Auckland
Private Bag 92019
Auckland, New Zealand.

Postal Address
Dr P.M. Fenwick
Dept of Computer Science
The University of Auckland
Private Bag 92019
Auckland
New Zealand.

E-mail p_fenwick@cs.auckland.ac.nz

Telephone + 64 9 373 7599 ext 8298

FAX + 64 9 373 7453

The Burrows–Wheeler Transform for Block Sorting Text Compression—Principles and Improvements

Abstract

A recent development in text compression is a "block sorting" algorithm which permutes the input text according to a special sort procedure and then processes the permuted text with Move-to-Front and a final statistical compressor. The technique combines good speed with excellent compression performance.

This paper investigates the fundamental operation of the algorithm and presents some improvements based on that analysis.

Although block sorting is clearly related to previous compression techniques, it appears that it is best described by techniques derived from work by Shannon in 1951 on the prediction and entropy of English text. A simple model is developed which relates the compression to the proportion of zeros after the MTF stage.

1. Introduction

A recent report by Burrows and Wheeler[1] describes a novel text compressor which permutes the entire input and then compresses that permutation. The authors state that their "algorithm achieves speed comparable to algorithms based on the techniques of Lempel and Ziv, but obtains compression close to the best statistical modelling techniques". Subsequent work by Fenwick[2] and Wheeler[3] has confirmed this statement; even though statistical compressors have had significant recent improvement, the original block sorting algorithm has also been improved. It continues to match all but the best statistical compressors in compression, while exceeding them in speed.

The compression technique is known by several names. Burrows and Wheeler initially described it as "block sorting". Later Wheeler used the terms "block reduction" and "block expansion" for the compression and expansion operations, and more recently Nelson [4] has

described the technique as the "Burrows-Wheeler Transform" (BWT). The term "Burrows-Wheeler Transform" appears to be gaining popularity and is included in the title of this paper, even though the term "block sorting" is retained throughout the text.

The BZIP routine of Seward[5] implements the ideas of this paper, using the "structured model" of section 6.2 and is released through the Free Software Foundation. The original version of BZIP had the same compression performance as the author's routines and will be used to illustrate block-sorting performance; the released version has a different final coder and slightly poorer compression.

The project described in this paper was intended to answer the following questions—

1. What are the principles which really underly the operation of the compressor, and how does it relate to other, better established, compression techniques?
2. It achieves good compression with a Huffman final coder, whereas the best compressors now use high-order arithmetic coders. What improvements might result from using these better coders?
3. Are there alternatives to the context-based sort or MTF which give better compression?

The first two questions are the main ones addressed in this paper, with the basic sorting and MTF operation taken "as given" after some initial experiments. Some alternative permutations and their coding have now been addressed by Arnavut and Magliveras[6].

During work on the block sorting algorithm, it became apparent that it is in many respects quite different from better known compressors. Not only does it require a new understanding of modelling techniques in its final compression stage, but it also differs at a much more fundamental level, appearing to be a representative of a "new" class of compression algorithms. This paper emphasises the philosophy underlying the operation of block sorting compression and includes the results of some compressors including the new modelling techniques.

2. The Block Sorting Transformations

The unique part of block sorting compression is two quite separate transformations, a

forward transformation which permutes the input data into a form which is easily compressed, and a matching reverse transformation which recovers the original input from the permuted data. The data is always considered in blocks which may be as large as the entire file or may be a few tens or hundreds of kilobytes. A block sorting compressor operates in three distinct stages —

1. An initial sort permutes the input text
2. The permuted text is processed by a Move-to-Front operation
3. The Move-to-Front output is processed by a statistical compressor.

2.1 The block sorting forward transformation

There are two approaches to describing the forward transformation. Burrows and Wheeler describe it as the steps –

1. Write the entire input as the first row of a matrix, one symbol per column
2. Form all cyclic permutations of that row and write them as the other rows of the matrix
3. Sort the matrix rows according to the lexicographical order of the elements of the rows
4. Take as output the final column of the sorted matrix, together with the number of the row which corresponds to the original input

In terms which are more familiar to workers in data compression, the transformation may be described as –

1. Sort the input symbols, using as a key for each symbol the symbols which immediately follow it, to whatever length is needed to resolve the comparison. The symbols are therefore sorted according to their *following* contexts, whereas conventional data compression uses the *preceding* contexts. (Some implementations do use preceding contexts, but the discussion is easier with following contexts.)
2. Take as output the sorted symbols, together with the position in that output of the last symbol of the input data.

2.2 The block sorting reverse transformation

The reverse transformation depends on two observations –

1. The transformed input data is a permutation of the original input symbols
2. Sorting the permuted data gives the first symbol of each of the sorted contexts

But the transmitted data is ordered according to the contexts, so the *n*-th symbol transmitted corresponds to the *n*-th ordered context, of which we know the first symbol. So, given a symbol ***s*** in position *i* of the transmitted text, we find that position *i* within the ordered contexts contains the *j*-th occurrence of symbol *t*; this is the next emitted symbol. We then go to the *j*-th occurrence of *t* in the transmitted data and obtain its corresponding context symbol as the next symbol. The position of the symbol corresponding to the first context is needed to locate the last symbol of the output. From there we can traverse the entire transmitted data to recover the original text.

2.3 Illustration of the transformations.

To illustrate the operations of coding and decoding we consider the text "*mississippi*" as shown in Figure 1. The first context is "*imississip*" for symbol *p*, the second is "*ippimissis*" for symbol *s*, and so on. The permuted text is then "*pssmipissii*", and the initial index is 5 (marked with "→"), because the fifth context corresponds to the original text.

	Forward transform		Reverse transform			
	symbol	context	Index	symbol	context	link
	p	imississip	1	p	i…	5
	s	ippimissis	2	s	i…	7
	s	issippimis	3	s	i…	10
	m	ississippi	4	m	i…	11
→	i	mississipp	5	i	m…	4
	p	pimississi	6	p	p…	1
	i	ppimississ	7	i	p…	6
	s	sippimissi	8	s	s…	2
	s	sissippimi	9	s	s…	3
	i	ssippimiss	10	i	s…	8
	i	ssissippim	11	i	s…	9

Figure 1. The forward and reverse transformations

To decode (right-hand part of diagram) we take the string "*pssmipissii*", sort it to build

the contexts ("*iiiimppssss*") and then build the links shown in the last column. The four *i* contexts link to the four *i* input symbols in order (to 5, 7, 10 & 11 respectively). The *m* context links to the only *m* symbol, and the two *p*'s and four *s*'s link to their partners in order.

To finally recover the text, we start at the indicated position (5) and immediately link to 4. The sorted received symbol there yields the desired symbol *m* and its immediately following context symbol, the fourth *i*. We then link to 11 get the *i*, and so on for the rest of the data, stopping on a symbol count or the return to the start of the file.

2.4 Move To Front coding

Most contexts permit only a few following symbols; if we collect together similar contexts so too do we restrict the symbols which may occur. MTF coding assumes that symbols can be ranked according to their "recency", or the closeness of their last occurrence. The compressor maintains a list of symbols; when a symbol is encountered its position or *rank* in the list is emitted as the value to the coder and the symbol is then moved to the front of the list. The effect is that more frequently-used symbols stay closer to the front of the list, while less frequently-used symbols drift to the back of the list. Smaller list ranks tend to be more frequent and can be emitted with shorter codes, while the less frequent larger ranks require longer codes.

Move-To-Front can itself be used as the basis for a text compressor, with a variable-length or statistical coder as an output stage. An example is the "Move-to-Front" compressor of Bentley et al[7] which uses words as the fundamental coding unit; in this paper we use 8-bit bytes as the unit of MTF coding.

3. Implementation and Results

The original BW94 implementation of Burrows and Wheeler used Huffman coding of the MTF output, with encoding of runs of rank-0 codes. The current work was done entirely with arithmetic coding because such coders were readily available and because it was felt that they should give better compression than Huffman coding. A version of block sorting was

implemented with a simple order-0 arithmetic compressor as its final stage (257 symbols and no historical or context information) and tested on the files of the Calgary compression corpus[1], with the results shown in Table 1. Following accepted convention, all compression results will be expressed as $^{output\_bits}/_{input\_bytes}$ (a value of 2.0 implies compression to 25% of the original size) and we compare compressors using a simple unweighted mean of the individual results.

Results from this compressor are included as "bsOrder0" in Table 1, together with "BW94" (the results from the original paper by Burrows and Wheeler). The standard when the BW94 algorithm was announced was Moffat's PPMC[8]. Comparisons with more recent compressors are given later. The newer bsOrder0 suffers in comparison with BW94 because it does not include run-encoding.

The two block sorting implementations compare well with PPMC for compression performance. Also in Table 1 are values from bsOrder0 showing the fraction of MTF output ranks which are 0; for most files the codes presented to the final coder are dominated by 0's, with more-compressible files having more zeros. Most text files give about 50% zeros, while the very compressible PIC has 87% zeros and the incompressible GEO has only 36%. A model developed in Section 9 relates the compression to the proportion of zeros.

File	size bytes	BW94	PPMC (1990)	bs Order0	frac 0 MTF
BIB	111,261	2.07	2.11	2.13	67%
BOOK1	76,8771	2.49	2.48	2.52	50%
BOOK2	610,856	2.13	2.26	2.20	61%
GEO	102,400	4.45	4.78	4.81	36%
NEWS	37,7109	2.59	2.65	2.68	58%
OBJ1	21,504	3.98	3.76	4.23	51%
OBJ2	246,814	2.64	2.69	2.71	68%
PAPER1	53,161	2.55	2.48	2.61	58%
PAPER2	82,199	2.51	2.45	2.57	55%
PIC	513,216	0.83	1.09	0.92	87%
PROGC	39,611	2.58	2.49	2.67	60%
PROGL	71,646	1.80	1.90	1.84	73%
PROGP	49,379	1.79	1.84	1.82	74%
TRANS	93,695	1.57	1.77	1.60	79%
AVG		2.43	2.48	2.52	63%

Table 1. Initial results on Calgary compression corpus

[1] The files of the Calgary compression corpus are available by anonymous FTP from `ftp.cpsc.ucalgary.ca: /pub/projects/text.compression.corpus/textcompression.corpus.tar.Z`

4. Improving the sort speed

Although it has no effect on the final compression, a crucial step in the compression algorithm is the sort phase to reorder the input text. While reasonable speed is obtained from a simple radix sort based on the first digraph of each context, the file PIC is not handled well—it has long runs of 0's and over 80% of it is placed in a single radix-sort bucket. Its runs often require comparisons of thousands of bytes to resolve, leading to very long execution times. These problems are solved by first run-encoding the input text[2]. Most files have their compression reduced by about 0.1%, but about 75% of PIC is absorbed in the runs and its compression improves by about 10%. More importantly, its sort time is reduced to about 2% of the original time. (Input run-encoding is also useful for PPM compressors.)

The number of comparison operations can be reduced by collecting 4 8-bit bytes into 32-bit words, striping bytes across successive words and striding across 4 words between word comparisons. Each raw comparison operation therefore compares 4 bytes, usually taking no longer than a simple 1-byte character compare.

In his later implementation, Wheeler[3], has a further improvement on the word-oriented sort. It uses a 256-way radix sort and initially stripes bytes across 32-bit words, as described above. The sort buckets (or "groups") are then sorted by size and processed smaller groups first. As each group is completed the low-order 24 bits of each newly-sorted word are replaced by its index in the sort group. This ensures that each sorted symbol is uniquely tagged and comparison strings which include it are resolved immediately.

5. Statistical Compressors

Statistical compressors rely on coders and coding models which are sensitive to the symbol frequencies and adjust their operation in accordance with those frequencies. Most compressors then adjust symbol frequencies according to their contexts so that most contexts have few preferred symbols and may be emitted with very short codes; this contextual analysis accounts for much of the complexity, execution time and compression performance of these compressors.

[2] This approach was discovered independently by Wheeler and the author

A different technique was employed by Shannon[9] in his classic paper on the information content of English text. In his experiments a symbol "predictor" determined a probable next symbol, which was accepted or rejected by a "comparator" which could see the incoming text. Shannon presented two alternative methods.

1. The response to an offered symbol is either "CORRECT", or "THE CORRECT VALUE IS …".
2. The predictor must keep suggesting symbols until the correct one is found. The effect is that the predictor ranks the symbols in order of decreasing likelihood and the sequence of NO's and the final YES constitutes a unary coded value of that ranking.

Shannon's second method replaces each symbol by its *rank* in a list ordered according to symbol likelihood. It is a recoding of the input symbols, with the recoding usually dynamic according to the symbol context. There is an output symbol for each input symbol, and compression relies on the very skew output distribution, with most output symbols being 0 (no error) and able to be emitted with very short codes. A "hybrid" method requires a few YES/NO responses, but then gives the correct answer. It recognises that ranking is difficult and unreliable after the first few symbols, but is also justified by results in the Appendix.

Several compressors may be seen as using symbol ranking, although few have actually used that term or recognised the connection with Shannon's work. Bentley et al[7] describe a Move-To-Front compressor, but with words rather than letters as units. Lelewer and Hirschberg[10] use self-organising lists to store the contexts in a compressor derived from PPM. Howard and Vitter[11] develop a compressor based on PPM, but one which explicitly ranks symbols and emits the rank. Much of their paper is devoted to the final encoder, using combinations of quasi-arithmetic coding and Rice codes. Their final compressor has compression approaching that of PPMC, but is considerably faster. A more recent paper by Yokoo[12] also describes a compressor based on these ideas.

6. Interpretation of Block Sorting

Cleary et al[13] showed that the permutation step of block sorting can be achieved using

the data structures of their PPM* compressor, so that block sorting is in a sense equivalent to context modelling compressors such as the PPM family. Context modelling compressors are already known to be equivalent to the dictionary compressors of Ziv and Lempel.

Thus despite the initial appearance of the block sorting algorithm being quite different from other data compression algorithms, it clearly relates to established techniques. Its underlying operation is however quite different. For example, quite apart from the initial permutation, it employs high-order contexts without emitting escapes to control the order in the decompressor.

6.1. An interpretation following Burrows and Wheeler

We can regard block-sorting compression as a sequence of three processes —

1. The initial, sorting, stage permutes the input text so that similar symbol contexts are grouped together. The permutation creates strong locality because the grouping of the (invisible) contexts collects together the few symbols likely to occur in each context.
2. The Move-to-Front phase then converts the various local symbol similarities into a single global similarity. The most likely symbol in each neighbourhood converts to a 0, the next most likely to a 1, and so on. Whereas the local contexts are fairly dynamic and fast-changing, the global one is much more stable with relatively constant statistics, even though it is at best an approximation to the true local contexts.
3. The final compression stage exploits the highly skewed frequency distribution from the second stage to produce efficiently-compressed output.

6.2. A symbol ranking interpretation

The author has developed an alternative interpretation which considers block sorting as a symbol ranking compressor. In a sequential symbol-ranking compressor each context has its own individual symbol ranking list and the compressor must maintain an appropriate complex of contexts and lists, switching between contexts as individual symbols are processed. In block-sorting compression the Move-to-Front list is used as the current estimate

of the symbol ranking. The initial sorting transformation, by collecting together similar contexts, also collects their similar symbol ranking lists. The similar rankings ensure that the preferred symbols usually change relatively slowly and that a single ranking list is adequate for all contexts.

The heart of the operation is therefore the Move-to-Front list, with the initial sorting transformation forming a preprocessor which ensures that the list remains close to its optimum ordering. In comparison with simple Move-to-Front (MTF) we see that it includes contextual information, whereas MTF works entirely from symbol recency — it may be most accurate to regard the sorting phase as a preprocessor for assisting a traditional MTF "compressor".

Sequential compressors such as PPM recognise explicit contexts and develop individual coding models for those contexts. With permuted input text the coder, and especially the decoder, have no precise knowledge of the contexts; changes in the emitted symbol are a very poor indicator of changes in context and we cannot infer context changes from the pattern of emitted symbols. However we can usually assume that the context for one symbol is very similar to that for the preceding symbol and that their symbol rankings are likewise similar. The ignorance of the precise context is irrelevant; we just assume that it resembles the one we have just handled.

7. Improved Final Coders

The incentive for this work came from realising that the original block sorting algorithm achieved good performance with a simple Huffman final compression stage, and that better results might be obtained with more complex statistical compressors. Accordingly, this paper deals only with adaptive arithmetic coders in the final stage.

One of the very early discoveries was that conventional "good" compressors gave quite poor performance and that new techniques were needed. Tests with a PPM compressor of variable maximum order gave the results of Table 2 for selected files of the Calgary Corpus; the "better" the compressor, the worse the results. It is clear that the output of the MTF

recoder has little of the context structure on which conventional compressors rely.

File	bs Order0	COMP4 order 2	COMP4 order 3	COMP4 order 4
BIB	2.02	2.17	2.29	2.38
OBJ1	4.01	4.59	4.64	4.67
PAPER1	2.51	2.75	2.92	3.31
TRANS	1.64	1.70	1.76	1.83

Table 2 . Compression with a conventional "good" compressor in final stage.

From measurements of the distributions of the code values to the final stage, it is conjectured that all non-zero values occur as random (Poisson) events against a background of rank-0 events. The final compression stage has no context information to exploit, as all of that has been absorbed by the initial sort phase. It must rely on local frequency effects.

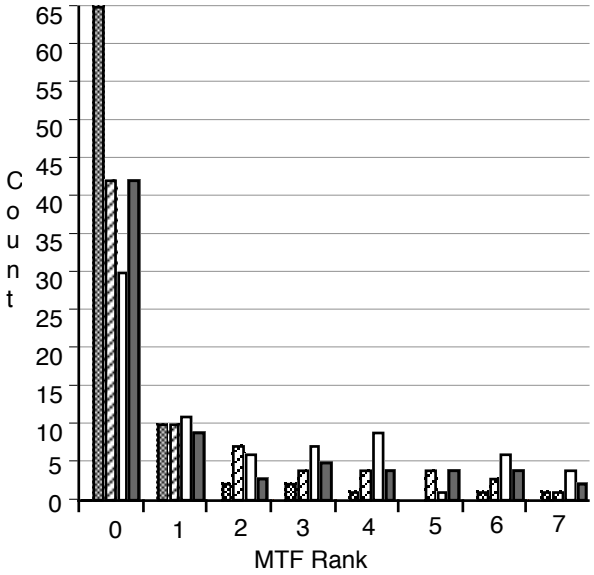

Figure 2. MTF counts in successive samples

Figure 2 shows the frequencies of the first few symbol ranks for successive 100 symbol samples of the file PAPER1. The proportion of 0's varies from 65% to 30% and the proportions of 4's from 1% to 8%, even over this short distance. Thus not only must we cope with a highly skewed distribution, but we must also handle one with significant short term variations.

A conventional arithmetic coding model is defined by its symbol alphabet, its total count

and its increment. Skewness requires a wide range of symbol frequencies, and a large ratio of total count to increment, while fast adaptation to local variations requires a small alphabet and a total count much closer to the increment to force frequent re-scaling.

These conflicting requirements are resolved by using a hierarchical system of arithmetic coders in which most of the fine structure is captured with small "foreground" models for the few most frequent symbols, with "background" models handling the less frequent symbols. The simplest such system uses a hierarchy of small "caches", each handling a few more-probable values and escaping to more-remote members of the hierarchy. For example, a 3-level model of {4-member cache, 16-member cache, full model} compressed the corpus to an average of 2.42 bit/byte, which is a useful improvement over 2.52 of the simple order-0 arithmetic coder. Better results were achieved with the models of the next two sections; both are hierarchical.

Conventional wisdom says that the hierarchy of models is not needed and that a single model will always adjust to the symbol statistics. That is true with a static symbol distribution, but here we are trying to respond to local variations in the frequencies and that demands coders with fast adaptation.

In all cases the arithmetic coding models are tuned by adjusting increment and maximum count. An increment of 16–32 and a total count of 8192–16,384 are reasonable first approximations for most models.

7.1. A unary coding model

Shannon's second method in Section 5 elicits a sequence of YES/NO responses which can be handled with a simple binary coding model. A single binary model gives an overall compression of 2.67 bits per byte, improving to 2.56 bits/byte with an order-1 context for the binary models. (If the last symbol was a 0, there is a reasonable chance of it being followed by another 0 of a run; a 1, in the middle of a unary-coded value, is quite likely to be followed by another 1.)

As the relative probability of YES and NO responses changes with successive NO's or

erroneous predictions of a symbol, it is advisable to change to a new (binary) arithmetic coding model after each prediction failure, reverting to the base model as we start each new symbol. The one set of order-0 coding models is shared by all symbols, with the coding probabilities being the symbol counts in each model. Thus each digit of the unary code should be handled by a separate binary coding model with its NO signalling an escape to the next model in the sequence. It is shown in the Appendix that after about 6 NO responses it is appropriate to emit the correct symbol instead of forcing more attempts. (The threshold is not at all critical.) This is the "hybrid" Shannon method from Section 5, and gives compression of 2.36 bit/byte on the Corpus.

7.2. A Structured coding model

A second coding model exploits the skewness of the distribution, dividing the alphabet of about 256 symbols into successive groups of 1, 2, 4, 8, … symbols, as shown in Figure 3. If the symbols had an exact $^{1}/_{\text{rank}}$ distribution (Zipf's law) they would be equally distributed across the entries of the first-level model; there is negligible benefit in adapting the model to the correct distribution. The model as described gives an overall 2.36 bit/byte on the Calgary corpus, the same as for the unary coded models.

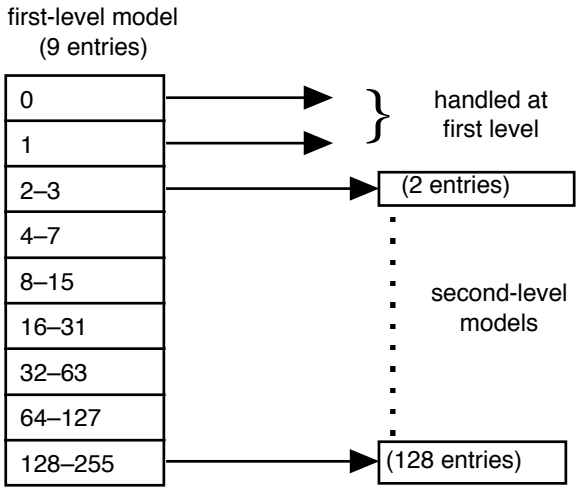

Figure 3. Structured coding model

7.3. Run-length encoding

As much of the MTF output consists of runs of 0's (correctly-predicted symbols) it is sensible to consider run-length encoding of these symbols. Unfortunately, even though the principles of run-encoding are well established, the efficient coding of runs of low-entropy symbols is still a difficult task. There are two approaches –

1. If a run of length N is encoded as an explicit count, coding the length requires about $\log_2 N$ bits, plus any overhead for signalling and terminating the run.
2. With an arithmetic coder it is possible to encode the "run" symbols directly and allow the coder to adapt to their high frequency. In considering a run of length N we must consider (N+1) symbols, N symbols for the run itself, each with probability $N/(N+1)$ and the terminator with probability $1/(N+1)$. With a fully-adapted arithmetic coder, coding the symbols of the run requires about $N \log_2 ((N+1)/N)$ bits, while the run termination (an improbable symbol) requires about $\log_2 N$ bits.

Both cases are dominated by the $\log_2 N$ term and the two approaches have a similar cost. Neither is clearly superior and this is borne out in practice. The chosen method allocates code values 0 and 1 to coding the bits of (run_length+1); all ranks greater than 0 are increased by 1 before coding. With the least-significant bit emitted first, the most significant 1 can be omitted as implicit in the terminating non-run symbol. (This method is equivalent to Wheeler's "2/4" length encoding[3].) Combining this run-encoding with the structured model gives an overall 2.34 bit/byte on the Calgary corpus, the best result for a block sorting compressor.

8. Performance

Table 3 gives the compression performance and speed for several compressors, all run on a 275 MHz DEC Alpha.

	Bits per byte						Time (seconds)			
	Unary coder	bred	BZIP	PPMZ	GZIP level-9		bred	BZIP	PPMZ	GZIP level-9
Bib	1.98	2.07	1.95	1.80	2.51		0.47	1.21	6.68	0.32
Book1	2.40	2.60	2.40	2.31	3.26		5.78	11.84	160.59	3.19
Book2	2.06	2.19	2.04	1.93	2.70		4.20	8.22	78.62	1.93
Geo	4.55	4.87	4.48	4.64	5.34		0.80	1.33	16.15	0.98
News	2.53	2.62	2.51	2.31	3.06		2.24	4.45	35.60	0.98
Obj1	3.93	3.91	3.87	3.72	3.83		0.10	0.20	6.87	0.07
Obj2	2.49	2.58	2.46	2.29	2.63		1.33	2.72	18.29	1.13
Paper1	2.48	2.58	2.46	2.29	2.79		0.18	0.51	2.88	0.14
Paper2	2.44	2.58	2.42	2.29	2.89		0.33	0.94	5.17	0.27
Pic	0.77	0.83	0.77	0.75	0.82		0.78	1.51	> 5 hrs	4.96
ProgC	2.52	2.58	2.50	2.32	2.67		0.13	0.37	2.14	0.11
ProgL	1.73	1.78	1.72	1.52	1.81		0.23	0.64	4.26	0.25
ProgP	1.72	1.78	1.71	1.57	1.81		0.15	0.47	3.38	0.21
Trans	1.50	1.56	1.50	1.28	1.61		0.34	0.86	4.68	0.20
AVG	2.36	2.47	2.34	2.21	2.70	**TOTAL**	17.06	35.27	345.34	14.74

Table 3. Summary of results : block sorting and other compressors

Unary Coder Block sorting with the Unary coding model of Section 7.1.

bred Wheeler's compressor of his 1995 report. It combines his fast sort algorithm with a Huffman output coder, avoiding the cost of a final arithmetic coder.

BZIP An early version of Seward's implementation combining Wheeler's best sorting techniques with the author's model of Section 7.2 and run-encoding as in Section 7.3. It is equivalent in compression to the best compressors of this paper.

PPMZ Bloom's recent implementation of PPM[14]; this is the best compressor known to the author. (Its time for compressing PIC is some hours, even on the fast computer used for these tests.)

GZIP the standard compressor of the GNU software release, operating at level-9 for best compression. At the default level it gives similar compression but takes about half the time.

Burrows and Wheeler state that their method combines the power of statistical compressors with the speed of Ziv-Lempel techniques. This is confirmed here, with bred being nearly as fast as GZIP, but with better compression. Its compression compares well

with the state of the art when block-sorting was originally announced (PPMC, 2.43 bit/byte, Table 1). BZIP obtains rather better compression, but the speed is halved by the more complex arithmetic final coder. BZIP is about 10 times faster than PPMZ, even with PIC omitted from the PPMZ results.

9. Compression and Rank-1 Probability

Figure 4 shows the variation of rank frequency with rank for three files of the corpus, plotted on log-log scales. The relative frequency of each rank is approximately of the form $prob \approx rank^{-2}$ (the dotted line in the diagram has the correct slope, but is moved vertically). More-compressible files have a greater slope, while less-compressible have a lesser slope. It is convenient here to use 1-origin numbering for the MTF ranks, instead of the more usual 0-origin numbering, to allow logarithms of the ranks.

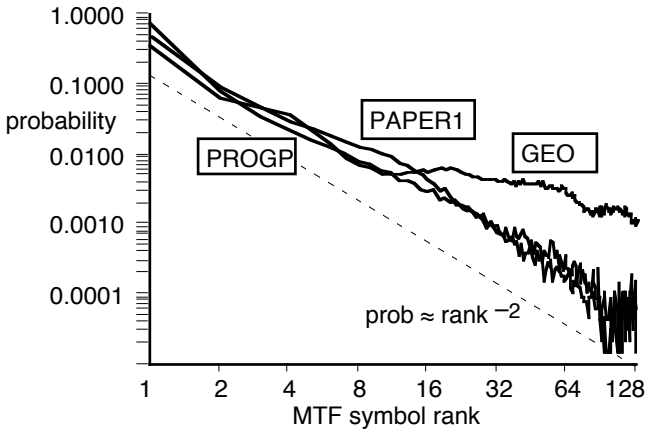

Figure 4. Probabilities of the first 128 MTF ranks for three files

We have already seen from Section 3 that more compressible files have more correctly predicted symbols. These observations can be used to construct an empirical model relating compressibility to the proportion of correct predictions. If we assume a first rank probability P_1 and that the probability of rank r_i is pr_i^a, we can adjust p and a to obtain a total probability of 1 over the input alphabet (usually 128 or 256 symbols) and then calculate the entropy of that code considered as a source.

Given the initial probability P_1, Table 4 shows the exponent required to force the total probability to 1, the consequent entropy for the files of the corpus, and with it the actual

compression of the files of the Calgary corpus with BZIP. A better view is in Figure 5, where we plot the predicted entropy (solid line) and measured values as a function of P_1.

File	frac 0 MTF	predicted power	predicted entropy	BZIP
BIB	67%	-2.19	1.92	1.95
BOOK1	50%	-1.71	3.06	2.40
BOOK2	61%	-2.00	2.30	2.04
GEO	36%	-1.39	4.11	4.48
NEWS	58%	-1.91	2.49	2.51
OBJ1	51%	-1.78	3.00	3.87
OBJ2	68%	-2.23	1.84	2.46
PAPER1	58%	-1.92	2.46	2.46
PAPER2	55%	-1.84	2.66	2.42
PIC	87%	-3.36	0.75	0.77
PROGC	60%	-1.98	2.33	2.50
PROGL	73%	-2.42	1.56	1.72
PROGP	74%	-2.47	1.49	1.71
TRANS	79%	-2.74	1.20	1.50

Table 4. Predicted entropies assuming a power-law distribution of rank frequencies

The fit is excellent, considering that some files show considerable divergence from a strict power law behaviour and that the values are based on only the probability of the rank-0 symbol. Three of the binary files (GEO, OBJ1 and OBJ2 – all labelled) are well above the line and less compressible than the empirical model might indicate. OBJ1 also suffers because it is too small to develop good context statistics, while the larger "book" files benefit from their larger contexts.

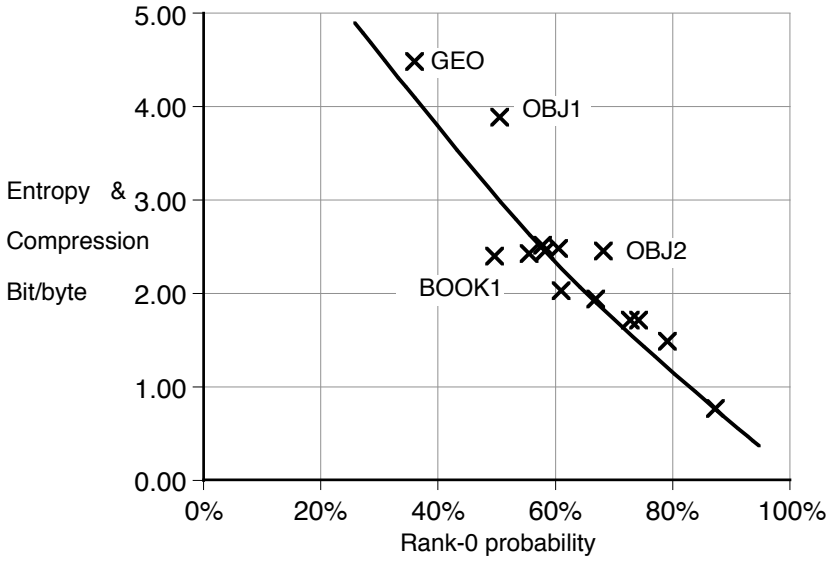

Figure 5. Code entropy v Rank–0 probability – comparison with experimental results

10. Conclusions

Block sorting data compression is an interesting and novel technique which has been confirmed as a practical data compression scheme, combining high speed with good compression performance. It is already established as a member of the general family of statistical compressors, being realisable in terms of data structures used with more-conventional statistical compressors. The crucial "block-sorting" initial step acts as a preprocessor to the already known Move-to-Front transformation. Together these two operations provide an implementation of a context-dependent "symbol ranking" transformation first described by Shannon in 1951. Although provably equivalent to PPM and similar statistical modelling techniques, block sorting is best regarded as a member of this long-established though little-recognised compression method. A simple model has been developed which shows good agreement with the observed compression on many files.

The output to the final statistical compressor has a very skew frequency distribution and little of the context structure on which normal compressors depend. Good compression then requires coders specially adapted to this data.

Some very recent work by Arnavut and Magliveras[6] may be regarded as a generalisation of the block sorting permutation. Their "Linear Permutation Sorting Algorithm" (LPSA) considers a cyclic group of permutations of order *n*. Communicating one of the group generators and a group exponent may allow a compact specification of a permutation which gives better compression than that provided by the block sorting permutation. The block sorting permutation is then a particular *ad hoc* permutation which is easily specified for both the forward and reverse permutations; while good it is not necessarily optimum.

Acknowledgements

This work was performed while the author was on Study Leave at the University of California–Santa Cruz, the University of Wisconsin–Madison and the University of Western Australia. Funding was provided by Research Grant A18/XXXXX/62090/F3414032 from the

University of Auckland. The author acknowledges the contributions of all of these institutions.

The assistance and ideas of Prof David Wheeler are especially acknowledged. He developed the original algorithm and provided many useful comments on this work. Other important contributions came from Mike Burrows and Charles Bloom.

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. (1996) "Block Sorting Text Compression", *ACSC-96 Proc. 19th Australasian Computer Science Conference, Melbourne Jan 1996*. pp 193–202
`ftp.cs.auckland.ac.nz/out/peter-f/ACSC96paper.ps`

[3] Wheeler, D.J. (1995) private communication. (Oct)
[This result was also posted to the `comp.compression.research` newsgroup. The files are available by anonymous FTP from `ftp.cl.cam.ac.uk/users/djw3`]

[4] Nelson,N., "Data Compression with the Burrows-Wheeler Transform", Dr. Dobb's Journal, September 1996, p 46.

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

[6] Arnavut, Z. and Magliveras, S.S. (1996) "Block Sorting and Compression", *IEEE Data Compression Conference, DCC'97*, Snowbird Utah

[7] Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei.,V.K. (1986) "A locally adaptive data compression algorithm", *Communications of the ACM*, Vol 29, No 4, pp 320–330.

[8] Moffat, A. (1990) "Implementing the PPM Data Compression Scheme", *IEEE Trans. Comm.*, Vol 38, No 11, p1917–1921.

[9] Shannon, C.E. (1951) "Prediction and Entropy of Printed English", *Bell System Technical Journal*, Vol 30, pp 50–64, Jan 1951

[10] Lelewer, D.A., Hirschberg, D.S (1991)., "Streamlining Context Models for Data Compression", *IEEE Data Compression Conference*, DCC-91, Snowbird Utah, pp 313–322.

[11] Howard, P.G., Vitter, J.S. (1993) "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding", *IEEE Data Compression Conference*, DCC-93, Snowbird Utah, pp98–107.

[12] Yokoo.H., "An Adaptive Data Compression Method Based on Context Sorting", *IEEE Data Compression Conference, DCC'96*, Snowbird Utah, pp 160–169.

[13] Cleary, J. G., Teahan, W.J., Witten, I. H. (1995) "Unbounded Length Contexts for PPM", *IEEE Data Compression Conference, DCC-95*, Snowbird Utah, pp 52–61.

[14] Bloom, C. (1996a), "LZP: a new data compression algorithm", *IEEE Data Compression Conference, DCC'96*, Snowbird Utah

Appendix. Unary v Entropy coding of ranks

For the first few symbol-rank values we could code the value as either a unary code or as an "entropy code", where the codeword length is $\log_2(1/P_i)$, and P_i is the probability of the *i*–th symbol. Arithmetic coding will function as an entropy code. Figure A1 shows the two codelengths against rank for a typical symbol distribution as encountered here. A simple unary code is more efficient than the entropy code for values of 6 or less.

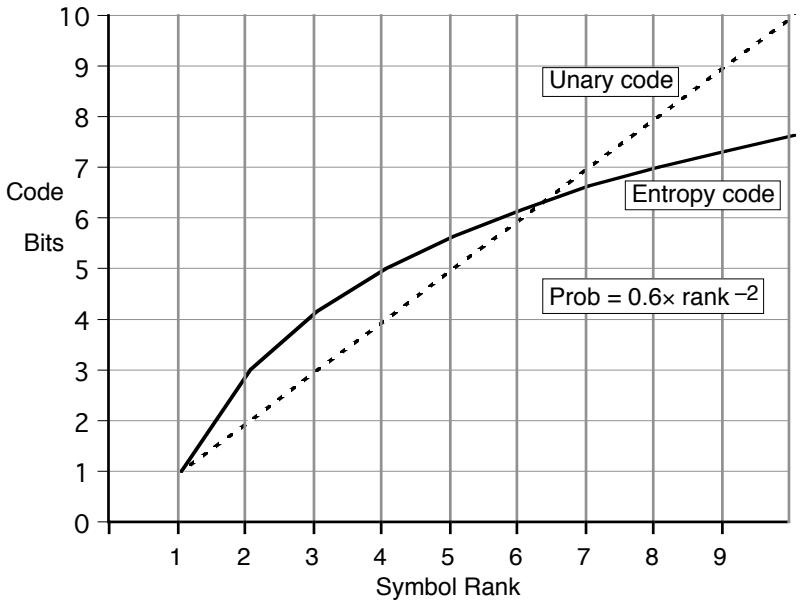

Figure A1. Unary and entropy coding of a source

The transition between unary coding and entropy coding is also justified by considering Huffman coding. If, when constructing a Huffman code and adding the *j*-th symbol of an *N* symbol alphabet with individual symbol probabilities $P(S_i)$, we have

$$P(S_j) > \sum_{j+1}^{N} P(S_i)$$

then the whole of the preceding sub-tree (bottom-up construction and representing the right-hand side) becomes one branch of the new tree and the new symbol becomes the entire other branch. More specifically, if

$$P(S_j) > 2P(S_{j+1}) \qquad \text{for all } j$$

or the probabilities form a geometric series of common ratio less than (–2), the Huffman tree is degenerate, as shown in Figure A2.

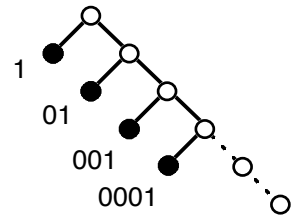

Figure A2. Degenerate Huffman tree

But we have already seen that the n-th symbol here has a probability proportional to n^{-b}, where b is about 2. Combining these two results, the Huffman coding tree is found to be degenerate for the first few most probable symbols, with those symbols represented as simple unary codes.