Burrows Wheeler Compression: Principles and Reflections

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

Abstract

After a general description of the Burrows Wheeler Transform and a brief survey of recent work on processing its output, the paper examines the coding of the zero-runs from the MTF recoding stage, an aspect with little prior treatment. It is concluded that the original scheme proposed by Wheeler is extremely efficient and unlikely to be much improved.

The paper then proposes some new interpretations and uses of the Burrows Wheeler transform, with new insights and approaches to lossless compression, perhaps including techniques from error correction.

1 Introduction to Lossless Data Compression

Lossless data compression involves the compression of files such that they can be later recovered bit-wise identical to the original. Comprehensive descriptions are in the book by Bell et al[3] or a more recent one edited by Khalid Sayood[18]. Several introductory matters must be introduced though, before any detailed discussion of any specific algorithms –

1. The "Calgary Corpus"[9] is a somewhat arbitrary collection of 18 diverse files (text, program and binary, and conventionally restricted to a subset of 14 files) that forms a de-facto standard for comparing lossless compressors. Results are usually quoted as "bits per character" (bpc) or the final compressed size (in bits) divided by the input size (bytes or characters) for each file; the 14 values are then averaged to give an overall measure or figure of merit. A "reasonable" compressor will give a corpus average of about 3.0 bpc, while the best approach 2.1–2.2 bpc.

2. The "workhorse" compression algorithm is "LZ-77", first described by Ziv and Lempel[24] in 1977 and especially as implemented in GZIP, WinZip and others. It uses the most recent 8k–32k bytes as a dictionary and replaces "phrases" (symbol sequences) by pointers to recent occurrences of those phrases. It is simple and fast with good but not excellent compression ('Calgary average' about 3.0bpc average).

3. The LZW variant[21] of LZ-78[25] is generally preferred in data communications. It builds a dictionary of recent phrases and emits indices of these phrases. LZW is usually faster then LZ-77 but gives less compression (average say 4.5 bpc).

4. Prediction by Partial Matching (PPM) includes the best compressors[10, 11]. A sequence of say 4 immediately preceding symbols is used as a "context" to the current symbol; as most contexts have few possible following symbols, these possibilities can be encoded in relatively few bits, especially if

the encoder recognises symbol probabilities. The compressor builds a dictionary of known contexts with the frequencies of their following symbols. Compression as low as 2.1 bpc has been reported. An important aspect of PPM is its use of "escapes". If a symbol does not exist in the current say order-4 context, the compressor emits an "escape" to a shorter or lower-order context which is probably less restrictive and includes more symbols. The handling of escapes and especially the estimation of the escape probability is *the* major difficulty with PPM.

5. Burrows Wheeler compression, the subject of this paper, also uses contexts but sorts all input symbols according to their adjoining contexts; the sort gives a permutation of the original data. It gives "PPM compression with LZ-77 speed", to about 2.25 bpc (2.34 bpc for the production implementation BZIP2).

Bell et al[3] show a formal correspondence between LZ-77 and PPM, as do Cleary and Teahan[11] between PPM and Burrows Wheeler.

2 Introduction to Burrows-Wheeler Compression

Burrows and Wheeler announced their "Block-sorting Lossless Data Compression Algorithm" in 1994[8] as a combination of three successive processing stages.

1. The initial stage is the "Burrows Wheeler Transform" (BWT) proper, which performs a context-dependent permutation of all of the symbols of the input data (or blocks thereof for large files). Because similar contexts usually adjoin similar sets of few symbols, the permuted data has extensive groupings of similar symbols and especially runs of single symbols (cf the symbols in PPM contexts).
2. A following recoding stage, usually Move-To-Front[6] (MTF) or recency, transforms the permuted output into a strongly skewed distribution of small integers, with a preponderance of zeros (often about 60%).
3. The final statistical coder performs an efficient coding of the second-stage output. This stage is usually either a dynamic Huffman coder or an arithmetic coder.

Considerable work has been published on improvements in each of these three areas. The author has presented early work[13], and a survey to about 2000[16]. The current state of the art is given by Abel, in a paper included in this volume[1]. The following sections of this paper describe the Burrows-Wheeler Transform itself, with brief discussions of topics 2 and 3 above, leaving detailed treatments to other authors in this issue.

2.1 The Forward Transform

The Transform proper can be performed either as a sort (which the author still uses) or as a suffix tree or variant. Most published work has been with suffix trees, improving either their speed or memory efficiency and sometimes with minor changes to the transform. But these improvements are largely irrelevant to the present work, which emphasises the post-transform processing; the transform is still the transform and different versions deliver similar output to later stages.

The forward transform is illustrated in Fig 1 using the traditional matrix description for the input string "`kaukapakapa`"[1], to give the permuted output {`ppkkkuaaaaa`, 8}. The three steps are –

[1] As a change from the usual *mississippi*, Kaukapakapa is a (small!) locality in New Zealand

	Rotated input	Sorted rotations	Final	
1	kaukapakapa	akapakaukap	p	
2	akaukapakap	akaukapakap	p	
3	pakaukapaka	apakapakauk	k	
4	apakaukapak	apakaukapak	k	
5	kapakaukapa	aukapakapak	k	
6	akapakaukap	kapakapakau	u	
7	pakapakauka	kapakaukapa	a	
8	apakapakauk	kaukapakapa	a	⇐
9	kapakapakau	pakapakauka	a	
10	ukapakapaka	pakaukapaka	a	
11	aukapakapak	ukapakapaka	a	

Figure 1: The Forward transform

1. Form all cyclic rotations of the input text (column 1)
2. Sort the rotations into lexical order (column 2). From the cyclic nature, the *start* of each row is the following context of its *final* symbol.
3. Transmit the *last* column as the transformed output (column 3) together with the index of the *original* text in the sorted rotations (here 8 and marked by ⇐)

3 The Reverse Transform

In principle the Burrows Wheeler Transform yields just another permutation of the original text. But is (apparently) unique in that the permutation itself contains all the information needed to recover the original with only a single integer, or extra sentinel symbol, needed apart from the transformed data. We give two versions of the reverse transform. Both rely on the fundamental properties that the k-th 'c' in the first column (sorted contexts) corresponds to the k-th 'c' in the last column (permutation) and the m-th symbol in the permuted data adjoins the m-th sorted context.

The first version, based on sorting, is perhaps more descriptive, but has a cost of order $\mathrm{O}(n^2 \log n)$ (n sorts, each of cost $n \log n$). The second and usual implementation, that used by Burrows and Wheeler, requires only 2 passes over the received data and has cost $\mathrm{O}(n)$.

3.1 Reverse transform by sorting

The sorting version, in Fig 2, recovers the complete rotation matrix by a sequence of sorts and concatenations. Observe that successive symbols of the permuted output "belong" with successive contexts. (The last column of each row (the output) is cyclically associated with the start of the same row (the context).) The order-1 contexts (column 2 of Fig 2) are obtained by sorting the permuted symbols (column 1) and are concatenated with matching permuted symbols to give context symbol pairs in column 3.

But each of these pairs is itself a context in its own right; these order-2 contexts, sorted, equally apply to the permuted input. So we sort the symbol pairs and prepend the permuted symbols to give the triples of column 5, which may in turn be sorted to give the order-3 contexts of column 6. Repeating this sequence builds the order-4, order-5, ..., contexts and eventually the order-n contexts shown here in column 10. This column is the same as the complete rotation matrix of Fig 1.

1	2	3	4	5	6	7	8		9	10
Ord 1		Ord 2		Order 3		Order 4			Order 11	
p	a	pa	ak	pak	aka	paka	akap	...	pakapakauka	akapakaukap
p	a	pa	ak	pak	aka	paka	akau	...	pakaukapaka	akaukapakap
k	a	ka	ap	kap	apa	kapa	apak	...	kapakapakau	apakapakauk
k	a	ka	ap	kap	apa	kapa	apak	...	kapakaukapa	apakaukapak
k	a	ka	au	kau	auk	kauk	auka	...	kaukapakapa	aukapakapak
u	k	uk	ka	uka	kap	ukap	kapa	...	ukapakapaka	kapakapakau
a	k	ak	ka	aka	kap	akap	kapa	...	akapakaukap	kapakaukapa
a	k	ak	ka	aka	kau	akau	kauk	...	akaukapakap	kaukapakapa
a	p	ap	pa	apa	pak	apak	paka	...	apakapakauk	pakapakauka
a	p	ap	pa	apa	pak	apak	paka	...	apakaukapak	pakaukapaka
a	u	au	uk	auk	uka	auka	ukap	...	aukapakapak	ukapakapaka

Figure 2: Reverse Transform, by sorting

In particular the 8th row is the original text (refer back to the arrowed row in Fig 1). Transmitting this index along with the permuted data allows the correct ordering to be selected. Alternatively, some authors terminate the string with a sentinel, often denoted by '$' and select the row corresponding to the sentinel in the permuted text. This approach is perhaps easier to describe, but is more awkward with binary files where all symbol values are legal.

3.2 Reverse transform by permutation

Remembering the two fundamental properties from Section 3, observe that the first symbol within the permuted data adjoins the *first* 'a' context[2]. The next symbol adjoins the *second* 'a' context and so on. Therefore, arrange links to chain successive 'a's of the permuted data, then successive 'b's, etc. These links allow each symbol to be associated with its following context, and that symbol with its context, as will be seen in Fig 4.

Figure 3 gives a complete procedure for performing the reverse transform, working from an input array `in[]` to an output array `out[]`. (Both input and output arrays start data at position 1.) Other parameters are the length of the data string (`size`) and the starting position or index of the original data in the transformed array (`start`).

The reverse transform operation is demonstrated in Fig 4, the left part showing the context information and the right the links to recover the original text. We enter at position 8 (`start` = 8), and immediately link to its context (k, at 5), and successively to a (at 11), u (at 6) and so on. At each stage deliver the permuted symbol as output. This sequence will eventually traverse the whole file, visiting every symbol, because of the cyclic rotations. Indeed each and every starting position yields a different cyclic rotation of the input; the integer (`start` = 8) simply selects the rotation corresponding to the original data. Because the input text loops back on itself, head-to-tail, there is no "natural" end to this cycle and we terminate after processing `size` symbols, or by returning to `start`. (More list traversals, similar in spirit but different in detail, may be seen later in Fig. 9 and in Section 7.2.)

The major costs are the initial scan to count symbols, and the final traversal to build the output, both apparently of $\mathrm{O}(n)$. A simple "big-Oh" estimation is misleading though, and really applies only to computers with a traditional uniform-access RAM. With the hierarchical or multi-level memories of modern computers the arrays are often too large to fit in caches near the processor. The final scan is essentially random through the arrays and gives many cache misses, with corresponding speed penalties.

[2]Assume for simplicity that the symbol alphabet is the letters {a, b, c, ..., z} with each letter occurring several times.

```
void reverseTransform(int size,     // length of data
            char in[ ],             // input data
            char out[ ],            // output data
            int start)              // starting position
    {
    int counts[maxChar],            // count of each symbol
        link[maxSize],              // link symbol to context
        ctxNext[maxChar];           // next symbol in each context
    int i, j, sym, ix;

    for (i = 0; i < maxChar; i++)   // clear counts
      counts[i] = ctxNext[i] = 0;

    for (i = 1; i <= size; i++)     // count input symbols
      counts[(int)in[i]] ++;

    ctxNext[0] = 1;                 // start at first symbol
    for (j = 1; j < maxChar; j++)   // get context starts
      ctxNext[j] = ctxNext[j-1] + counts[j-1]; // over prev context

    for (i = 1; i <= size; i++)     // set up links
      {
      sym = in[i];
      link[ctxNext[sym]] = i;       // copy next into link
      ctxNext[sym] ++;              // "consume" this occurrence
      }

   ix = start;                      // start of traversal
   for (i = 1; i <= size; i++)
     {
     ix = link[ix];                 // step to symbol's context
     out[i] = in[ix];               // copy context symbol
     if (ix == start) break;        // alternative termination
     }
    out[i+1] = 0;                   // optional C-string terminator
    } // end reverseTransform
```

Figure 3: C-procedure for reverse transform

symbol	first	count
a	1	5
k	6	3
p	9	2
u	11	1

(a) context limits

position	symbol	context	link	
1	p	a	7	
2	p	a	8	
3	k	a	9	
4	k	a	10	
5	k	a	11	
6	u	k	3	
7	a	k	4	
8	a	k	5	⇐
9	a	p	1	
10	a	p	2	
11	a	u	6	

(b) symbol links

Figure 4: The reverse transform by permutation

4 The Second-Stage Recoder

Most workers have followed Burrows and Wheeler in using variations of Move To Front in the second, recoding, stage. The main exceptions are Arnavut et al[2] using "Inversion Coding", Wirth[22] who eliminated the recoding stage completely in favour of PPM techniques and more recently Ferragina et al using Wavelet trees[17].

The second stage MTF[6] or recency coder maintains a list of all symbols; the current symbol is recoded into its position in the list and then moved immediately to the head of the list, forcing all intervening symbols back by one position. The resulting "working set" of active symbols near the head of the MTF list can be recoded into small integers, to be compactly encoded by the final statistical encoder. (From Fig. 1 observe the runs in the permuted sequence "`ppkkkuaaaaa`". All symbols of a run, except the first, recode to 0 with MTF.)

We would like symbols to enter the working set as soon as they are needed (if not just before the first usage!), remain only for as long as they are active and then disappear immediately. While the Move To Front or recency coder is a good approximation to this behaviour, its basic problem is the inherent asymmetry of the MTF operation itself. It is very strong at activating new symbols, but equally weak at forgetting them; they are brought immediately to the head of the MTF list, but then drift slowly back only as they are superseded by later symbols. Symbols which are genuinely in the current working set are often given more expensive codings on their next usage because the list is "polluted" by more-recent but not-to-be-reused symbols. Thus, which symbols should be retained as likely, and which might be better forgotten?

Ideally the MTF list should be changed whenever the context changes, but determining relevant context changes is hard enough for the compressor and even more difficult for the decompressor, with only partial information available to it. Most improvements to the MTF operation depend largely on examining the relative quietness or noisiness (local entropy changes) of the MTF output, and looking at the reuse statistics of symbols. However recent work by Deorowicz[12] and Ferragina[17] presents some effective solutions to the problems of determining and using context changes.

In general, the modifications try to rearrange symbols near the head of the MTF list, promoting those that seem likely to used in the near future and demoting those whose reuse seems less likely. As a comprehensive current survey is presented in a paper by Abel[1] in this issue, no further discussion is given here.

5 The Final Coder

The permutation of the Burrows Wheeler Transform and the MTF recoding perform no compression whatsoever (apart from run-compression in some implementations), although the skewed symbol frequencies from the recoding mean that that output is eminently compressible. Most of the responsibility for good compression lies with the final statistical encoder, although certainly assisted by the MTF or similar recoder.

Many BW compressors use a dynamic Huffman or similar final coder and Fenwick[15] even shows that respectable compression is possible using a static Universal or Variable-Length integer coding; compression results with variable-length codes are shown later in the column headed "VLcode" of Table 3. But for the best compression it is necessary to use an adaptive arithmetic coder, and that will be assumed henceforth.

The final statistical coder must reconcile two conflicting requirements.

- The MTF values have an enormous range of frequencies, typically greater than 10000:1. For good

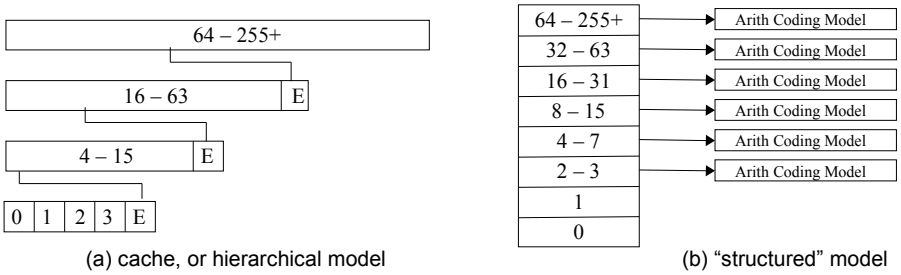

Figure 5: Two final coding models

coding the frequencies in the arithmetic coder model must cover at least this range of values. A typical coder with adjustable symbol increment and total model frequency requires the ratio

$$\frac{\text{total frequency}}{\text{symbol increment}} > 10000$$

- The most frequent symbols, precisely those which must be coded most efficiently, are subject to significant changes in frequency over quite short distances. An arithmetic coder is sensitive only to the ratio of the counts in its coding model; except for small changes when an active count is incremented, any significant changes to the ratios follow only from the model rescaling which reduces the relative significance of the less-active counts[3]. An agile coder, able to respond to rapid changes in relative frequencies, needs frequent rescaling and in turn a small ratio of model limit to symbol increment.

These requirements are usually reconciled by using complexes with small agile foreground arithmetic coders handling the frequent changes of small integers from the MTF head and larger background coders for the occasional references to the larger values from the MTF tail. In his very early examination of Burrows Wheeler compression Fenwick[13] introduced two special combinations of arithmetic coders, both shown in Fig. 5. Both coders allow for alphabet expansion beyond values 0–255, to handle run-encoding and End-Of-File signalling.

- The "cache" model of Fig. 5(a), is entirely analogous to a virtual memory hierarchy, hence the name. It emits all small MTF values from a small coder with an "escape" (cf. virtual memory 'miss' or page fault) to a larger model for larger values and similarly to higher levels. Another typical model (different from that of Fig 5, because the coding ranges are subject to considerable optimisation.) might encode 8 values in level 1 (0–6, ESC), 32 values at level 2 (7–38, ESC) with the rest (39–127 or 39–255, EOF) at level 3.
- The "structured" model of Fig. 5(b) effectively brings all escapes into the first-level coder, emitting them as alternatives rather than in sequence as with the cache model.

In practice there is negligible difference between the two approaches, and the choice is largely one of personal preference. Final coders are surveyed by Abel[1] and Deorowicz[12].

6 Examination of Run encoding

There has been considerable work on both the MTF recoding mechanism after the BW Transform and the final statistical coder, reducing the Calgary average from 2.40 bpc for the original BW94[8] to around 2.25

[3]Most arithmetic coders have a maximum total count for all symbols within the model, with all symbol counts halved whenever this total is exceeded.

bpc for the very latest results, an improvement of around 6.7%. But this improvement is often expensive in processing especially where there is extensive rearrangement of the MTF list, or calculation of coding statistics.

The MTF output is dominated by zeros (often about 60%) with many occurring runs. Many workers (starting with Wheeler) encode these runs in an effort to further compress the output. However there seems to have been rather little critical analysis of the handling of these runs that predominate the permuted and recoded data. Thus while most authors use run-encoding, and some recognise it as an example of a universal code, few say much more about the actual encoding of the run length or discuss alternative ways of encoding the run length as an integer. A closer look at run-encoding is the main emphasis of the next section of this paper.

(Run length encoding is sometimes used before the transform, initially to accelerate the sorting operation, especially of the Calgary file `pic`. However it usually has little effect on the overall compression and is not needed with suffix tree implementations of the transform. Deorowicz[12] and Abel[1] discuss some other placements of run length encoding.)

6.1 Post-transform run compression

The output of the MTF (or similar) recoding has two alternating components, the "noisy" MTF data and the "quiet" runs of zeros. As 50–60% of the whole output stream is of zeros, contributing perhaps 20–30% of the output bits, efficient coding of these runs is crucial to the efficiency of the compression.

A frequent coding of zero-runs is that used by Wheeler[23] and is stated by him as being of unknown origin. Quoting from his report –

> *This works by extending the character set by one value. We increase other codes by 1 and use 0 or 1 to represent groups of zero as follows. The first 0/1 gives 1/2 zeros. The second 0/1 gives 2/4 zeros. The third 0/1 gives 4/8 zeros etc.*

Alternatively, encode (`length`+1) as a binary value (emitting the digits 0 and 1 as the symbols 0 and 1, least-significant digit first) but with the most significant 1 omitted, to be implied by the next "non-run" symbol. (Wheeler's 1/2, 2/4, ..., interpretation is sometimes easier for *decoding*.) Some typical codes are shown in Table 6, with 'x' and 'y' being any non-zero values.

An important property, noted from the very first use with BW compression, is that this coding *never* expands the file. An isolated 0 is still recoded as a single symbol, but all longer runs are decreased in length by the coding. Remember though that the alphabet is extended by one symbol and that most symbols are encoded as a larger value. This may slightly reduce the compression at this stage.

Input stream	length	output code
$x0y$	1	$x0y$
$x00y$	2	$x1y$
$x000y$	3	$x00y$
$x0000y$	4	$x10y$
$x0000000y$	7	$x000y$

Figure 6: Examples of Wheeler's run-length coding

Wheeler also describes a more general version of this code, for repeating arbitrary characters, rather than zeros.

- A run is recognised as two successive identical symbols, say binary value c.
- Encode the bits $\{0, 1\}$ of the binary run length with the values $\{c, c \oplus 1\}$ ($\oplus$ is the exclusive-OR). Any symbol other than c or $c \oplus 1$ terminates the count and implies its most-significant 1.
- Insert an extra $c \oplus 2$ if the run is followed by either $c \oplus 1$ or $c \oplus 2$.

6.2 General comments on encoding the run-length

The run length is a "variable-length" or "universal" code embedded in a data stream and each coded value must contain three elements – some introduction or key to signal the start of the embedded code, an encoding for the value, and a terminator or some other way to determine the length.

As with most universal codes, the smaller values are the most frequent and special care must be taken in their encoding. It is relatively easy to design a code which is efficient for large values, but much harder to make one which is good for the smallest values *and also* transforms smoothly into one which is efficient for large values. The efficiency of coding large values (long runs) is relatively unimportant, because the infrequent long runs represent many symbols and almost any reasonable representation will provide considerable benefit.

6.3 Run-length coding; details for one file

Table 1 shows the coding details for the Calgary file 'bib', as a representative text file, for the more frequent MTF codes. The "position=0" column refers to the symbol counts *before* any run-compression; compare the counts here with those later in Table 2. Two thirds of the transformed symbols are zero, contributing 22% of the final bit stream. Even after run-compression, the zeros contribute twice as many bits as the next most frequent code. Any improvement in run-encoding is likely to give a useful improvement in compression.

Table 1: MTF output details for "bib"

MTF position	0	1	2	3	4	5	6	7
Bytes	74297	10042	4478	3259	2542	1989	1797	1536
Bits	47507	23643	16490	13327	11351	9452	8815	7777
Bit/Byte	0.639	2.354	3.682	4.089	4.465	4.752	4.905	5.063
Frac Bytes	66.8%	9.0%	4.0%	2.9%	2.3%	1.8%	1.6%	1.4%
Frac of Bits	22.0%	11.0%	7.6%	6.2%	5.3%	3.0%	4.1%	3.6%

111261 input bytes, 215893 output bits (1.940 bit/byte)

Table 2 shows details of the actual arithmetic coding at the *root* of the final structured coder. (All except the first two columns escape into branch coders, whose performance is unimportant here.) Within the final statistical coding the digits of the run length are encoded at 2–3 bits per digit, which seems rather inefficient for coding binary values, given that a simple binary coding model should be able to code at about 1 bit per digit. We therefore investigate some other ways of encoding the run lengths.

But first, Figure 7 shows the frequencies of short runs for several files. It is the general picture that is important, rather than the finer details, or any differences between the files. The distribution is highly

Table 2: Coding details for "bib"

Value range	0	1	2–3	4–7	8–15	16–31	32–64	64–257
MTF values	0-run lengths		1–2	3–6	7–14	15–30	31–63	63–256
Bytes	13039	7082	14520	9587	7886	3658	1220	94
Frac Bytes	11.7%	6.4%	13.1%	8.6%	7.1%	3.3%	1.1%	0.1%
Rt-Bits	27266	20241	28445	24238	21404	13577	6369	801
Root bpc	2.091	2.858	1.959	2.528	2.714	3.711	5.216	8.432
Fract of Bits	12.6%	9.4%						

These results use the "structured coder" of Fig. 5(b)
"Rt-bits" is the number of bits emitted by the "root level" of the structured coder
"Rt-bpc" is the number of bits emitted by the coder for each group

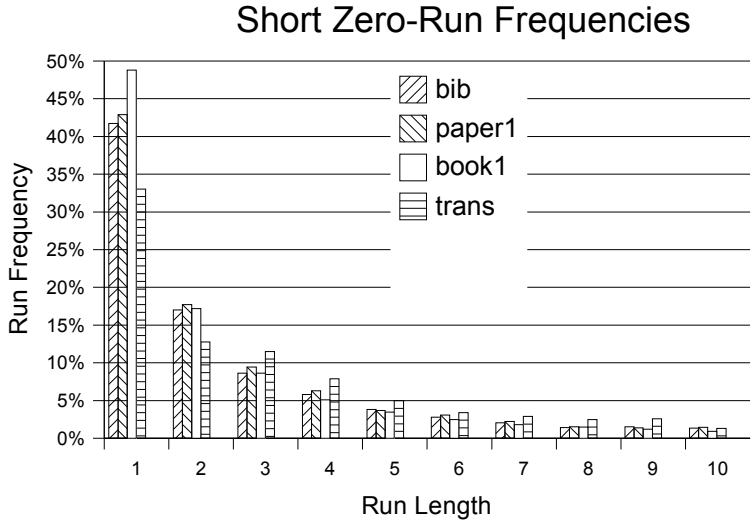

Figure 7: Run-length frequencies for some files

skewed, with 30–50% of the "runs" being solitary values, and 12–17% being pairs, much like the MTF codes themselves. The lesson is that short runs, and especially isolated zeros, must be coded very efficiently.

Conventional universal or variable-length codes[16] such as Elias' gamma and delta codes or Fraenkel's Fibonacci codes use only 0s and 1s with special combinations of bits or other formatting to both delimit each representation and encode its value. But these codes are inappropriate here where the 0s and 1s of the length representation are embedded within symbols from a much larger alphabet. Note, for example, how Wheeler introduces the length with a significant 0 or 1 which can only be part of the length coding and terminates the length with a "normal" data symbol which cannot be part of the length, but *also* implies a significant 1.

Two alternative run-length encodings were tested, both drawing on the author's experience with variable-length codes. The first ("Direct") was a very simple one, included as a "worst-case" reference, while the second was an attempted "better-effort" using experience from this code, and others.

6.4 A simple direct binary model

An initial test compressor recognised a run as starting with a single 0 and then coded the run length itself with a simple arithmetic coder. This coder handled values up to either 20 or 96, with larger values (longer runs) handled by emitting a 0 as an escape code into a general purpose binary value encoder used elsewhere in the compressor. The results, shown in Table 3, were not good, prompting a re-examination of the coding. This led in turn to a recognition of requirements for the embedded length code and some subtle features of the original Wheeler code. (The results are compared using the author's BW2000 as a reference; the file `geoR` is a byte-reversed version of `geo`[4]. Also included are results for the author's variable-length coder[15] mentioned in section 5, using the Elias gamma code for text files and Fraenkel's Fibonacci code for binary files. The original paper has full descriptions of these codes.)

Table 3: Experimental run-length encoders

File	Wheeler	Direct Arith Code		2-bit code		VLcode
	(BW2000)	Limit 96	Limit 20	2 digit	3 digit	
bib	1.940	2.013	2.019	1.980	1.961	2.168
book1	2.366	2.436	2.440	2.397	2.385	2.843
book2	2.023	2.097	2.104	2.062	2.044	2.332
geoR	4.268	4.261	4.262	4.288	4.287	5.380
news	2.476	2.561	2.562	2.516	2.504	2.744
obj1	3.813	3.867	3.870	3.835	3.820	4.124
obj2	2.442	2.515	2.520	2.482	2.463	2.593
paper1	2.450	2.547	2.554	2.491	2.473	2.687
paper2	2.400	2.484	2.490	2.439	2.422	2.722
pic	0.752	0.765	0.765	0.759	0.757	0.850
progc	2.489	2.595	2.601	2.536	2.515	2.692
progl	1.707	1.814	1.821	1.755	1.737	1.847
progp	1.710	1.826	1.835	1.768	1.744	1.823
trans	1.496	1.601	1.610	1.549	1.528	1.579
AVERAGE	2.309	2.384	2.389	2.347	2.331	2.599

6.5 A 2-bit (base-4) run-length encoding

The Wheeler code expands the alphabet by one symbol to make room for the extra code required for the number representation. What might be the consequences of expanding by another two symbols, giving 4 non-data codes and allowing the length to be encoded in base-4, or as bit pairs? This code followed from a recognition of the problems with the "direct" code of the previous section and was an attempt to design a more efficient code, in particular one that represented more short runs by only a single digit. Codes of 1 or 2 base-4 digits can represent all lengths up to 20 (4 of one digit and 16 of 2 digits) as shown in Table 4.

As before, longer runs are represented by an Escape code (here 33_4, replacing a length of 20) into a general binary coder. The results are shown in the "2 digit" column of Table 3. Again, there is no improvement over the original Wheeler code, although the average is much closer than with the simple direct model. Table 3 also includes an obvious extension to a 3-digit (6 bit) code which can handle lengths up to 83. (We can handle 83 values with 6 bits because of the implicit length from the following symbol.) This is better than

[4]Because the Calgary file `geo` gives better compression with *preceding* contexts rather than *following* contexts, some workers reverse the byte order of *all* binary files before compression. But Fenwick[16], shows that the difference is an artifact of the file `geo` itself, rather than binary files in general.

Table 4: Base-4 codings for short runs (2-digit version)

data stream	run-length	code
$a0b$	1	0_4
$a00b$	2	1_4
$a000b$	3	2_4
$a0000b$	4	3_4
$a00000b$	5	00_4
$a000000b$	6	01_4
$a0000000b$	7	10_4
$\cdots$	19	32_4
$\cdots$	20	33_4

code 33_4 used as escape

its 2-digit version, but still inferior to Wheeler's code. We concluded that these codes were unlikely to give significant improvement over Wheeler's original code and that further development was not justified.

6.6 Pressure of Runs

Balkenhol et al [4, 5] comment upon the "pressure of runs", a phenomenon affecting almost any history-dependent MTF or coder algorithm, and one also mentioned by other writers. Consider a "well-adapted" binary arithmetic coder encoding a run of zeros (the 1 may be an escape into a coder for the other symbols). After a run of N zeros, the estimated probability of a 1 will tend to $\frac{1}{N}$ and a 0 to $1-\frac{1}{N}$. The cost of coding each zero tends to $\log_2 \frac{1}{N}$, and the cost of the termination to $log_2 N$, precisely the cost of specifying the length of the integer giving the run length. But this same adaptation has also increased the costs of *every* following symbol (initially to $\log_2 N$) until the binary coder eventually recovers from the run. This is the main justification for run-encoding.

In a similar way, a lengthy succession of zeros may upset the statistics of any adaptive MTF coder or final statistical encoder unless the design allows for runs. It is probably best to recognise the run, allow for it of course in the code generation, but ignore it in the statistics generation. This is in line with a recommendation of [5].

6.7 Comments on the Run-Length coding

Table 5 compares the various run-length codings, but it is useful to note separately the advantages of the original Wheeler code.

- The (Wheeler) character which signals a run (0 or 1) is always part of the encoded length and can never occur in the main data stream. In the binary test program, the length coding always adds to the cost of the signalling 0.
- The binary length encoding must be terminated or contain some indication of length. In a separate binary coding of an N bit value, this can be expected to add a further $\log_2 N$ bits to the output bit-stream. Wheeler has the following symbol, which can never be part of the run encoding, implying the termination. The termination, and final 1, are therefore sent at no cost, except for the cost of expanding the symbol alphabet.

Table 5: Comparison of length Encodings

Wheeler	Direct	Base-4 Coding
Introduction to length		
Length is introduced by the first digit (0 or 1) of the encoded length count. There is no immediate overhead.	Length code is introduced by a zero value, which is not part of the encoded length, adding an overhead of one.	Length is introduced by a non-data code (0–3) which is the first digit of the length. No immediate overhead.
Coding of short runs		
Single zeros (length=1) are encoded as a single 0, with no additional cost. Pairs of zeros are encoded as a digit 1, saving one coded value.	All codes include the overhead of the initial 0 and then have the arithmetically-coded length added to this.	Short runs (1–4) are encoded as a single digit; longer runs (5–19) as a pair of digits. (And possibly runs to 83 as a digit triple.)
Coding of long runs		
All run lengths are encoded as 0/1 digits; the code naturally handles runs of any length.	Long runs (of low probability) escape to a "standard" coder for binary integers.	Runs of 20 (or 83) or longer escape to a binary integer code, as for the "direct" code.
Termination of length code		
The next "non-run" symbol ends the run *and* supplies the most-significant 1.	The length termination is implicit in the arithmetic length coding for short runs. For longer runs it is implicit in the fall-back integer coding.	Irrelevant, except that the non-run symbol indicates whether a 1-digit or 2-digit code. Longer runs as for the direct coding.
Other matters		
All non-zero symbols have their codes increased by 1. The larger alphabet causes a slight decrease in coding efficiency.	— — —	All non-zero symbols have their codes increased by 3. The larger alphabet gives a larger decrease in coding efficiency than for the Wheeler code.

- Wheeler has rather few additional bits to send – the least significant is always stated by the signalling code and the most significant bit implied by the termination. The "efficient" binary coding must send this bits explicitly. While long runs do need more code digits with the embedded code, moderate coding inefficiencies do not matter much because the long runs are relatively rare (see Fig 7). It is the short and frequent run lengths that must be optimised.

The base-4 code was designed following a critical examination of the problems with the simple arithmetic coder. It might be improved by a better integration with the final coder (such as encoding digits 0–3 directly in the base model). But it still suffers from the extension of the active alphabet by about 2% for text files, with a corresponding compression penalty. But much more seriously, it also expands the working set by 2 symbols and this is a much greater proportion of an effectively smaller (working set) alphabet. In summary any benefit of the better run coding is offset by the effect of the larger alphabets.

6.8 Run-length Coding—Conclusions

We have briefly examined much of the earlier processing associated with the Burrows Wheeler transform and compression, leading into a discussion of run compression and run-length coding as an aspect that has received little previous attention. The conclusion is that Wheeler's original method for encoding runs of zeros is an excellent choice, and one that is unlikely to be much improved. Indeed the whole design of the original compressor by Burrows and Wheeler seems, in retrospect, to be a remarkable combination of well-matched elements[5]. Later improvements have been slight, and often difficult.

The rest of this paper arose from general thoughts about the Burrows Wheeler algorithm and trying to develop it in quite different ways. The author, having stated[16] that the Burrows Wheeler Transform was poorly understood, attempted to develop some different approaches and understandings; none of these alternatives was very successful but they are presented here in the hope that they may inspire further work. The first alternative comes from considering the general nature of transforms, and the second from possible combinations of Burrows Wheeler with PPM compression.

7 Comments on Transforms

Much of physics depends on viewing problems from two or more aspects, each one giving a particular insight into an otherwise difficult situation. Simple examples include the particle/wave duality of quantum theory, conversion between rectangular and polar coordinate systems, and trigonometric functions and their inverses.

Similarly, physics and mathematics often *transform* problems from one "space" into another space, again to facilitate computation or provide a complementary view. An important example is the Fourier Transform, which converts between *time space* and *frequency space*. The Burrows-Wheeler transform similarly transforms from *text space* to *context space*. Unfortunately the transformation into context space loses all explicit knowledge of the actual contexts, which is precisely the knowledge that PPM uses to such advantage. The expectation of this work was that we might be able to recover information on the true context of each symbol and use that information to assist compression.

[5] Much of this quality may be a tribute to the experience of Wheeler who, starting work in 1948, was certainly one of the world's most experienced programmers.

7.1 A compressor using derived contexts

In contrast to most previous Burrows Wheeler work, we concentrate on the *reverse* transform, breaking it open and incorporating much of the reverse transform into the compressor. The overhead is that we must encode the counts of all symbols, sending information that is usually extracted at an early stage of the reverse transform. As most text files have an alphabet of about 80 symbols, each requiring 10–12 bits to encode the symbol and count, the overhead is probably around 1000 bits per file (say 2500 bits for binary files). (Because we know the frequency of each symbol, we can also build a simple model for each order 1 context and know when to clear it at the start of the next context.)

1. Encode into the compressed file the frequencies of all symbols in the source alphabet. This operation duplicates (or anticipates) the *first* step of the reverse Burrows Wheeler transform, the step that usually follows immediately after the recovery of the permuted symbols.
2. As the transformed text is processed, build links for each symbol to its context, now anticipating the *second* stage of the standard reverse transform.
3. Eventually the separate links will start to combine, building longer source fragments that can be used as contexts to enable better prediction of symbols. (Although we do not yet know the *positions* of these fragments within the text, they are all valid contexts; their positions do not matter.) The resultant structure is a *random graph*, where each node has an order of at most 2, and ultimately all nodes are linked into a single cycle.

The growing knowledge of the source can be used to guide selection of the likely symbols, increasing the efficiency of the recency or MTF process; this is the approach used in recent work by Deorowicz[12], whose "exhumed contexts", produce excellent results. The approach here is different, completely replacing the MTF recoding by direct coding, analogous to PPM, using the inferred source to predict probabilities of the known symbols and control the statistical coder.

7.2 An example of context recovery

Table 8 gives an example of context recovery, using the string `kaukapakapa`. The first two columns show the usual sorted cyclic rotations and then the permuted symbols, with column 3 showing the contexts. Column 4 gives the original text, showing how the links develop. A new link between two symbols is shown with an '=', while an existing, older, link is shown as a '−'. The final column shows the contexts that are discovered or added at each stage.

The first step is to transmit all of the symbol counts, allowing the order-1 context boundaries to be predefined. Within each order-1 context the symbol occurrences are encoded as a simple bit vector, this generally costing less than escapes to an order-0 context. We can thus construct an exact order-1 coding model for each context, exact at least as far as symbol *occurrence* is concerned, but not symbol *frequency*. For later discussion these order-1 contexts are conveniently called *major contexts*.

In more detail the processing proceeds as –

- At line 1, the first symbol `p` is known to link to the first `a`. With the usual reverse algorithm we just create a link to facilitate the final traversal when all symbols have been processed. Here, we can create an explicit occurrence of symbol `p` within a major context `a`, shown by `p{a}`. (These links can be created without knowing the future symbol; it is only its context that is important and we know that from the symbol counts.

	Rotations	sym	ctx	Links in original text	Recognised contexts (to order-3)
1	akapakaukap	p	ak...	`k a u k a p=a k a p a`	p{a}
2	akaukapakap	p	ak...	`k a u k a p-a k a p=a`	p{a}
3	apakapakauk	k	ap...	`k a u k=a p-a k a p-a`	k{a}
4	apakaukapak	k	ap...	`k a u k-a p-a k=a p-a`	k{a}
5	aukapakapak	k	au...	`k=a u k-a p-a k-a p-a`	k{a}
6	kapakapakau	u	ka...	`k-a u=k-a p-a k-a p-a`	u{k}, u{ka}
7	kapakaukapa	a	ka...	`k-a u-k-a p-a=k-a p-a`	a{k}, a{ka}
8	kaukapakapa	a	ka...	`k-a u-k-a p-a-k-a p-a=`	a{k}, a{ka}
9	pakapakauka	a	pa...	`k-a u-k-a=p-a-k-a p-a-`	a{p}, a{pa}, a{pak}
10	pakaukapaka	a	pa...	`k-a u-k-a-p-a-k-a=p-a-`	a{p}, a{pa}, a{pak}
11	ukapakapaka	a	uk...	`k-a=u-k-a-p-a-k-a-p-a-`	a{u}, a{uk}, a{uka}

'=' denotes a new link; '−' an existing or prior link
'$x\{yz\}$' denotes symbol x with following context yz

Figure 8: An example of context recovery

However, it is only the leftmost of a pair of linked symbols that is known explicitly; the right one is inferred from the containing context. This means that the very last symbol of a linked string, such as the final '`a`' of `u{ka}` on line 6, is never linked explicitly. The preceding symbol (here the '`k`') must be marked as being followed by an '`a`' to complete the known context. These context symbols are held in an array, parallel to the symbols. (While we know that the '`k`' is certainly followed by *some* 'a', we do not as yet know *which* '`a`'.)

- Lines 2 to 5 follow the actions of line 1 to complete the creation of major context `a`. (These order-1 contexts are strictly not needed because that information is already encoded.)
- Apart from creating `u{k}`, step 6 links to an already connected symbol, allowing creation of `u{ka}`.
- Line 7 introduces another action, where a new link is created *between* two existing links, shown as `p-a=k-a`. It looks as though apart from creating `a{k}` and `a{ka}` as before, we can now go back in the links and create the longer contexts `p{ak}` and `p{aka}`. But this is not the case. The backward links exist only because the two symbols associated with each link are both processed already; their contexts will never be visited again and there is no point in generating them now. The only contexts to generate and save are those for the current symbol, the one followed by "=" in Table 8.
- We have seen how an identified sequence '`abcde`' defines the right or following context `a{bcde}`; it clearly defines also the left or preceding context `{abcd}e`. Left contexts may be generated in parallel with right contexts.

A symbol must be processed using the context generated from the *previous* symbol, as the best approximation to the current context. It is only after the symbol has been processed (encoded or decoded) that its context can be created. (Remember that a compressor may use only information that it has sent to the decompressor.) Because we generate and use contexts only within the current major context there is never any use for contexts in earlier major contexts.

7.3 Compression results with derived contexts

An initial implementation used just the order-1 contexts and did not build the links and higher orders[6]. A later version (much later!) built a dictionary of all contexts as they were found, using the links as described above.

[6]The order-1 results were presented at the DIMACS workshop, Aug 2004

Table 6: Order=1 and order-4 context compressors

	BW2000	Wirth	order 0	order 1	left 4	right 4	Deorowicz
bib	1.926	1.990	2.133	2.188	2.616	2.245	1.887
book1	2.356	2.330	2.523	2.557	3.322	2.552	2.264
book2	2.012	2.012	2.198	2.245	2.933	2.288	1.953
geo	4.268	4.390	4.812	5.272	5.594	5.608	4.129
news	2.464	2.487	2.677	2.775	3.309	2.817	2.397
obj1	3.765	3.811	4.227	5.127	5.493	5.420	3.692
obj2	2.433	2.514	2.710	2.953	3.333	3.097	2.411
paper1	2.439	2.492	2.606	2.759	3.243	2.823	2.390
paper2	2.387	2.424	2.571	2.666	3.241	2.712	2.329
pic	0.753	0.743	0.919	0.892	1.077	0.960	0.714
progc	2.476	2.518	2.666	2.853	3.336	2.912	2.422
progl	1.697	1.763	1.839	1.994	2.456	2.056	1.660
progp	1.702	1.792	1.821	2.044	2.476	2.117	1.666
trans	1.488	1.622	1.601	1.831	2.220	1.891	1.451
Average	2.298	2.349	2.522	2.724	3.189	2.821	2.240

As described earlier, the first step is to transmit a table of the symbols (about 80–90 for most text files) and the frequency count for each, to enable simulation of the first stage of the reverse transform, before all permuted symbols have been read. Each context then starts with a bit vector, giving the actual symbols within the context and costing about 120 bits for most text files.

Results are shown in Table 6 and compared with some other compressors, including both the simpler order-1 and an enhanced order-4 compressor, all compressing the Calgary corpus –

BW2000 This the author's best compressor, using "sticky MTF" and the structured final coder[16]. While not the best it is reasonably simple and easily understood.

Wirth Wirth's compressor [22] applying PPM techniques to the permuted symbols. This is one of the best results not using the MTF recoding.

order0 A very early result from testing the BW algorithm, using MTF and a simple order-0 arithmetic coder. This is effectively a baseline from which most other BW compressors developed (at least those using an arithmetic final coder).

order1 The new compressor, but using only order-1 contexts.

order4 An order-4 implementation of the full mechanism as described here. Results are included for both left contexts ("left 4") and the more natural right contexts ("right 4" and on which most of the discussion was based).

Deorowicz This is the very recent result of Deorowicz[12]. It combines some of his earlier coding enhancements with a context-based MTF assistance to give the best Burrows Wheeler result to date.

Although the left contexts certainly achieve compression it is not even as good as that with right contexts, largely because a leftward context of one symbol is not a good predictor of the next symbol, the two contexts being less-well related than the corresponding right contexts. Experiments, ultimately unsuccessful, included a "mix-and-match" of various combinations of the two context directions. Thus neither of the derived context techniques was at all successful; they were poorer even than the simplest "order-0" compressor!

Two points are crucial for compression performance; we must be able to handle most symbols with the most efficient methods and we need to limit the choice of likely symbols at each position. Neither succeeds here. Detailed compression statistics show that –

- Although bytes are compressed quite efficiently at order-4 (0.5–0.6 bpc), there are rather few of them, in fact often 70% of the symbols fall through to the "last resort" order-1 compressor. And most of the easy symbols are handled at the higher orders anyway, leaving the order-1 model to handle the more-difficult and expensive coding.
- The PPM style of compression must consider the complete ensemble of symbols within each context. It neglects the grouping of symbols that occurs in "classic" BW compression from the neighbouring higher-order contexts and which is captured by the MTF or recency mechanism. In effect the derived contexts have a much weaker detection of locality than is achieved with conventional Move-To-Front.

The combination of these two points means that no compression level is able to operate under optimal conditions, high orders because they can handle few symbols and order-1 because it is left only the more difficult ones. We conclude that purely context-based compression, while it certainly works, may be of no benefit as a replacement for the MTF stage.

This result may be contrasted with Deorowicz[12] who showed that recovered or "exhumed" contexts can certainly assist the MTF operation, especially when combined with an already-good final statistical encoder.

8 Combining PPM with Burrows Wheeler compression

PPM (Prediction by Partial Matching) has long been the best method for lossless or text compression, as introduced in Section 1. The handling of escapes is a major problem with PPM. Every context must allow for an escape but the escape probability is unknown and must be estimated. Many of the variants of PPM (PPMA, PPMB, PPMC[10], PPMD, PPM*[11] and PPMZ[7]) differ largely their prediction of the escape probability; increasingly accurate predictions of the escape probability give correspondingly better compression. An excellent coverage of escape handling is given in an unpublished report by Bloom[7], leading up to the extremely good results of his PPMZ compressor.

In this work we eliminate escapes completely by using a modified Burrows Wheeler transform to determine all possible contexts and the symbols within each, transmitting this context information as an initial part of the compressed data.

8.1 Use of Burrows Wheeler methods

We take a different view of the Burrows Wheeler transform, noting that –

1. The Burrows Wheeler transform is not of itself a compression algorithm (although with suitable post-processing its output is eminently compressible).
2. The Burrows Wheeler transform is rather a method of analysing the context structure of the input, with the permuted data presenting this information in a compact and convenient form.
3. The inverse Burrows Wheeler transform can recover all the original contexts from the permuted string, one for each symbol in the permuted data. (We usually select only that context corresponding to the original input, but the other contexts are all available.)

Normally, Burrows Wheeler comparisons may proceed to the full length of the input, to match the length of the context that we must recover. But we can limit the comparison length to say 4, so considering only

order-4 contexts. (With fixed-length contexts it may be more properly a Schindler Transform[19], rather than a Burrows Wheeler Transform.) Applying the standard reverse transform from each symbol position, but to only 4 symbols in each case will then yield all contexts of order-4; the frequencies of the following symbols then produce exact and complete coding models for each context.

Because the Burrows Wheeler transform performs a full context analysis of the entire file, we have full details of each context and its possible symbols and no escapes are needed if these models are used by a PPM coder.

8.2 Burrows Wheeler transform with PPM compression

We start with a description of the Burrows Wheeler transformation, using the input text `kaukapakapa`, but now modified as needed for PPM with preceding contexts (strings compared right→left and to a constant comparison length.

Forward Transform Take the input string and write all of its cyclic rotations to form a square matrix. Sort the matrix rows into increasing lexicographic order, but using *right-to-left* string comparisons to produce preceding contexts. For an order n compressor sort only the last n symbols of each row and then, if necessary, sort by the *first* symbol of the row, to produce symbol runs within each context. Emit the symbols in the first column of the permuted matrix.

Reverse Transform We start the reverse transform in the usual way, knowing that the permuted text can itself be permuted to produce the final letter of the contexts and producing a vector list of symbol indices. Traversing this list from any starting position gives a corresponding rotation of the input text or, equivalently, the preceding context of that symbol. Thus we can produce a context for each and every symbol of the original text.

The rightmost columns of Figure 9 show the recovered contexts (which are seen to be identical to the originals). Each context is shown here preceded by a continuation of the list traversal, showing that it is usually possible to generate contexts beyond the order of the Burrows Wheeler analysis. (These extended contexts, shown in italics, are almost incidental rather than a deliberate part of the algorithm.)

The final column shows the indices of the successive permuted symbols used in forming each context, reading from right to left. The approximate sorting means that most traversals end in a cycle and each illustrated traversal is long enough to show the cycle. The recovered context is accurate up until the symbol just before resuming the cycle. These longer contexts will be examined later.

The transmitted context information is the sequence `ppukkaaaaak`, which is essentially the normal Burrows Wheeler permuted output. But the limitation of context length and ordering by permuted symbol means that the sequence may be more compressible than normal Burrows Wheeler because we maximise symbol runs within each context. But it must still be encoded efficiently and any of the extant methods may be used.

Table 7: PPM processing

Context	ka	au	uk	ka	ap	pa	ak	ka	ap	select {p,u}
Symbol	p,u	k	a	p,u	a	k	a	p,u	a	in cols 1,6,7

To illustrate the PPM operation, assume that we already have the order-2 context `ka`, corresponding to the first two symbols, as shown in Table 7. (The first *order* symbols are transmitted "in clear", without compression.) The context `ka` contains the two symbols {`p,u`}; code must be emitted to resolve the uncertainty. The encoded `u` combines with the last symbol(`a`) of the context `ka` to produce the next contexts `au`, and then

sym-bol	context	Index	sym-bol	link	context ctx.sym	traversal left←right
p	pakapakau ka	1	p	6	ka.p	6-1-9-4-6
p	pakaukapa ka	2	p	7	*a* ka.p	7-2-10-5-7
u	ukapakapa ka	3	u	8	ka.u	8-3-11-8
k	kaukapaka pa	4	k	9	*a* pa.k	9-4-6-1-9
k	kapakauka pa	5	k	10	*a* pa.k	10-5-7-2-10
a	aukapakap ak	6	a	4	*p* ak.a	4-6-1-9-4
a	apakaukap ak	7	a	5	*p* ak.a	5-7-2-10-5
a	apakapaka uk	8	a	11	*a* uk.a	11-8-3-11
a	akapakauk ap	9	a	1	*k* ap.a	1-9-4-6-1
a	akaukapak ap	10	a	2	*k* ap.a	2-10-5-7-2
k	kapakapak au	11	k	3	*k* au.k	3-11-8-3

Figure 9: The forward transformation and recovered contexts

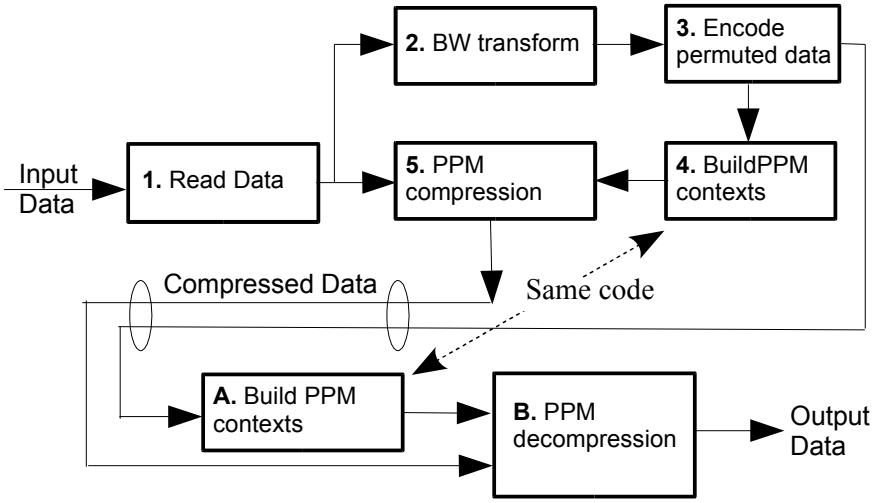

Figure 10: Data Flow within Compressor and Decompressor

uk, both deterministic and needing no output for the only possible symbol. The context ka has 2 possible symbols, needing an emitted code to select the symbol, and so on.

In this example, most of the encoding cost is in the context information (the Burrows Wheeler information), with very little in the PPM coding, as many symbols are in unique contexts and need no transmitted code.

8.3 The complete Burrows Wheeler PPM compressor

The flow of data within the complete compressor/decompresssor is shown in Figure 10. Steps 1, 2 and 3 at the top correspond to a conventional Burrows Wheeler compressor (except for the restricted context order), with the conventional BW compressed output forming the context description part of the compressed output. The context descriptions are then used to form constant-order PPM contexts (step 4), which are finally used (step 5) to generate the data part of the compressed output.

The decompressor first reads the context descriptions and then forms the PPM contexts (step A, using the same code as step 4 of the compressor). It then applies these contexts to the PPM encoded data (the second

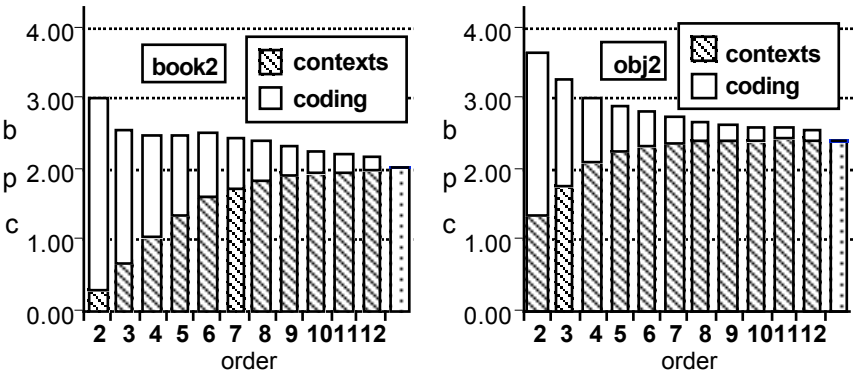

Figure 11: Compression cost vs context order for two files

part of the compressed output) to recover, at step B, the original data.

Figure 11 shows the costs of the two components, BW contexts and PPM compression, for two representative files from the Calgary corpus. The rightmost element of the histogram shows the compression for a reasonably good BW compressor. It was obvious from the outset that the context cost should increase with increasing order, while the PPM compression cost would decrease; it was hoped that the combined variations would produce a minimum (best compression) at some moderate order. but this minimum does not occur for *any* of the Calgary files.

But what these results *do* show is the relative costs of establishing contexts and of the PPM compression *per se*. Most of the cost lies in establishing contexts, especially at the high orders, so that for order 10 almost all files are performing the PPM compression at better than 0.3 bpc (bits per character) with many at less than 0.2 bpc. In many files the context information is about 95% of the compressed output. As this context information corresponds approximately to the escapes of standard PPM, it is easy to see why good escape management is so important to PPM compression.

8.4 Further work—the two context orders

Although the results are not as good as expected, they have indicated several possibilities for future work. The preceding discussion has spoken of the "the order" but there are actually two context orders involved –

- The Burrows Wheeler transform compares contexts to a precision given by the *BWorder*.
- The PPM contexts are recovered from the Burrows Wheeler output to a precision given by the *PPMorder*.

The *BWorder* and *PPMorder* have been assumed to be identical, but experience shows that a given *BWorder* can often generate rather longer *PPMorders*, as illustrated by the italicised contexts in Figure 9. Here contexts are extended by at most one symbol, but results from other files indicate that much longer contexts are often recovered with little extra work.

As good compression combines a low *BWorder* and high *PPMorder* we should investigate to what extent these combinations are possible. Experience in other areas of computer science indicates that while it is expensive to generate all of the contexts all of the time it may be much easier to generate most of the contexts and then supplement the existing (but low order) BW "primary contexts" with descriptions of the relatively few "secondary contexts" needed to complete the full suite of contexts for higher-order PPM.

8.5 Further work—techniques from Error Correction Codes

Is compression somehow related to error correction? Consider Shannon's original prediction/correction method for estimating the entropy of printed text[20], as implemented in the author's "Symbol Ranking" compressors[14]. The compressor and decompressor have matching predictors which try to predict the next symbol (probably from its context and the agreed previous data) and are in effect linked by an implicit zero-capacity information channel, supplemented by the visible "corrections channel". Any necessary correction to the initial predictions is sent as the compressed output over the corrections channel; the compressed data is in effect coding to recover from errors in the error-prone transmission through the implicit "data" information channel.

In this interpretation a Burrows Wheeler compressor simply assumes that each symbol is identical to its predecessor, with the MTF list holding symbols ranked in some estimated likelihood. The output codes are effectively corrections to this assumption; a larger MTF value corresponds to correcting a more serious prediction error.

A clearer example comes from PPM compression. Any symbol which cannot be resolved at the highest order is usually handled by an escape mechanism. If, instead of escapes we simply emit an unknown or "erasure" symbol we get an output similar to the output of an erasure channel, with occasional "corrupted" symbols. Figure 12 shows an excerpt from "The Prince" by Machiavelli, with all erasures as '?'. The relatively few "errors" are mostly isolated within considerable surrounding context.

```
When?ver those states which have been acquired as s?ate? have
been acc?stomed to live under their own la?s a?d in freedom?
there are three ?ourse? for those who w?sh to hold them? the
first ?s to ru?n them, the next i? to re?ide ?here in person?
the third ?s to pe?mit ?hem to live under their own laws,
?rawin? a tri?ute? and
```

Figure 12: Example of code with erasures, from "The Prince", by Machiavelli

Thus is it possible to *defer* the decision on these erasure codes until much more is known about their entire context, without relying on just the preceding context? Might *following* contexts give more assistance than the usual *preceding* contexts of PPM and therefore better compression? (The preceding contexts have been tried already and found inadequate; hence the escape or erasure.) Useful guidance may come from error-correction techniques for convolutional codes such as Trellis or Viterbi codes (but these suffer from severe combinational explosion with large alphabets), or sequential decoders such as the Fano algorithm. Work is proceeding in this area.

8.6 Burrows Wheeler with PPM—Conclusions

This final section has described investigations into a novel combination of a Burrows Wheeler Transform with PPM compression. Although the results are not as good as with standard Burrows Wheeler or PPM compression, they do clearly indicate the relative costs of the context information compared with the com-

pression *per se*. In most files, at reasonable context orders, at least 90% of the cost lies in establishing the contexts.

Two suggestions have been made for future work, one exploiting the ability to generate many contexts of rather greater length than that at which they were generated, and others suggesting the inclusion of error correcting coding methods into compression.

References

[1] Abel, J. *This volume*

[2] Arnavut, Z., "Move-to-front and Inversion coding", *Proc. IEEE Data Compression Conference*, March 2000, IEEE Computer Society Press, Los Alamitos, California pp193–202

[3] Bell, T.C., J.G. Cleary and I.H. Witten 1990. *Text Compression* Prentice Hall, Englewood Cliffs NJ. The Calgary Corpus can be found at ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[4] B. Balkenhol, S. Kurtz, "Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice", *IEEE Trans. on Computers*, Vol 49. No 10, October 2000.

[5] Balkenhol, B, Shtarkov, Y M. "One attempt of a compression algorithm using the BWT." SFB343: Discrete Structures in Mathematics, Faculty of Mathematics, University of Bielefeld, Preprint, 99–133, 1999, URL (October 2005): `http://citeseer.ist.psu.edu/balkenhol99one.html`.

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

[7] Charles Bloom, "Solving the Problems of Context Modeling", *informally published report*, see `http://www.cbloom.com/papers/`

[8] 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`

[9] The Calgary corpus may be downloaded from
ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[10] Cleary, J.G. Witten, I.H. "Data compression using adaptive coding and partial string matching", *IEEE Trans Communications*, COM-32, vol 4, pp 396–402, 1984.

[11] Cleary, J.G. and W.J. Teahan, 1997. "Unbounded length contexts for PPM". *The Computer Journal*, Vol 40, No 2/3, pp 67–75, 1997.

[12] Deorowicz, S., "Context exhumation after the Burrows Wheeler transform", *Information Processing Letters*, Vol 95, No 1, pp 313–320, 2005.

[13] 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.

[14] Fenwick, P.M., 1997. "Symbol Ranking Text Compression with Shannon Recodings", *J.Universal Computer Science*, February 1997.

[15] Fenwick, P.M., "Burrows Wheeler Compression with Variable Length Integer Codes", *Software Practice and Experience*, Vol 32, No 13, Nov 2002, pp 1307–1316..

[16] Fenwick, P.M., "Burrows Wheeler Compression", *Lossless Compression Handbook*, Ed Khalid Sayood, Academic Press, 2003, pp 169–193.

[17] P. Ferragina, R. Giancarlo, G. Manzini. "The myriad virtues of wavelet trees". International Colloquium on Automata, Languages and Programming (ICALP), 2006, pp 560–571.

[18] *Lossless Compression Handbook*, Ed Khalid Sayood, Academic Press, 2003.

[19] Schindler, M., 1997. "A fast block sorting algorithm for lossless data compression", *IEEE Data Compression Conference*, March 1997, IEEE Computer Society Press, Los Alamitos, California, p469

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

[21] Welch, T.A. "A technique for high-performance data compression", *IEEE Computer*, Vol 17, No 6, pp 8–19, June 1984.

[22] Wirth, A.I., and A. Moffat, "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.

[23] Wheeler D., Description of the program 'bred', private communication

[24] Ziv, J. and Lempel, A. "A universal algorithm for sequential data compression", *IEEE Trans Information Theory*, Vol IT-23, No 3, pp 337–343, May 1977

[25] Ziv, J. and Lempel, A. "Compression of individual sequences via variable-rate coding", *IEEE Trans Information Theory*, Vol IT-24, pp 530–536, September 1978.