Improvements to the Block Sorting Text Compression Algorithm

Peter Fenwick

Technical Report 120

ISSN 1173-3500

3 August 1995

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

peter-f@cs.auckland.ac.nz

Abstract.

This report presents some further work on the recently described "Block Sorting" lossless or text compression algorithm. It is already known that it is a context-based compressor of unbounded order, but those contexts are completely restructured by the sort phase of the compression. The report examines the effects of those context changes. It is shown that the requirements on the final compression stage are quite different from those in compressors of more conventional design.

The report then presents several different approaches to improving the compression performance, eventually yielding a compressor which is among the best so far presented and is actually based on fundamental work by Shannon in 1950.

It is shown that the block-sorting technique compares well with other compressors in terms of compression factor, compression speed and working memory.

This report is available by anonymous FTP from

`ftp.cs.auckland.ac.nz /out/peter-f/report120.ps`

1. Introduction.

In a recently published report, Burrows and Wheeler [4] described a new text compression technique. That technique was examined in a recent report [6], and this present report extends that work with a view to improving the compression. (That earlier report [6] will be referred to as "Report 111" throughout this one.) Report 111 described the general technique of block-sorting and especially the algorithms for permuting the input file and then recovering the original text from that permuted file. An initial implementation was presented, with a simple order-0 arithmetic compressor as the final, compression, stage. Report 111 then presented the probability distributions of the symbols as submitted to the final compression stage and briefly discussed the effect of those distributions on compression.

Initially as a sideline to the main theme, Report 111 presented a new type of compression coding (actually dating back to a paper by Shannon in 1950 and subsequently overlooked!) and showed its relation to the block-sorting technique. The report demonstrated that the coding used by Shannon is competitive with others for handling the sorted data with MTF processing.

In this report we develop those ideas, firstly with techniques which exploit the distribution of symbol probabilities, and then with techniques based on the Shannon coder. One of the problems with block sorting is that the sorting phase is very slow for files with long runs of symbols, or repeated sequences. Some techniques were suggested for overcoming those problems and those are investigated here.

Underlying all the work is a realisation that, while block-sorting does use contexts of unbounded order, those contexts are reorganised so that conventional context-based methods are of little real help. (And that means virtually *every* extant text compression method!) Existing techniques must be adapted to the new environment.

2. Questions & Answers.

The earlier Report 111 was distributed to several people, who raised some useful questions. It is appropriate to answer those questions here as they illustrate some important aspects of the compression technique. It is done in a rather informal style. The questions are presented anonymously, with minimal editing.

Q *Why should the blending of the distributions (p. 16) be smooth? e.g. for random input, the order -1 distribution should carry the most weight; similarly for the context "q" in English text, the order 1 distribution is most important - all else should have much lower weight.*

A Why should they be other than smooth? It is not a good answer, except that large samples do tend to give smooth distributions. I do not like arguing from special cases, such as "q" in English, as they too easily lead to bad conclusions. In any case, random input does give an asymptotically smooth distribution – equal for all symbols! Is the argument for a "smooth distribution" any less plausible than other justifications for escape probabilities?

Q *You probably thought of this already, but I wondered why the context of a symbol is restricted to the following symbols (or, symmetrically, the preceding symbols). It seems possible that a two-sided context would give better compression, although slightly slowing down the sorting phase. More precisely, for a string of symbols ..., s_-2, s_-1, s_0, s_1, s_2, ... consider the context for symbol s_0. In the TR you take s_1, s_2, ... but it could be s_1, s_-1, s_2, s_-2, ... I think that decoding should work as before. There are obvious variations on this idea.*

A minor point - when sorting, do you include the symbol s_0 in the context ? Figure 1 implies that you don't, but elsewhere it seems that you might.

A Compression folk-lore has it that there is little difference between using the preceding context and following context. This was reported by Shannon in 1951 and most recently by Burrows and Wheeler. Anyway, why should the compression of a file be changed by processing it in reverse order?

At one stage I had the idea that given a symbol "*s*" with following context "*tuvw*…", we should sort on the key "*tsuvw*…" to bring together all occurrences of "*s*". It doesn't work, because the decoder can't reconstruct the correct context ordering. I also had the idea of doing a second block sort to collect similar symbols, but that doesn't work either. It is effectively an attempt to encode the output with a following context-sensitive compressor, and we have seen that that is not useful.

The symbol itself is not part of the context. Using the "surrounding" context (combining both preceding and following) would probably give little benefit, but lead to enormous complications in the decoding process.

Shortly after the above was written, Charles Bloom told me that he had *actually tried reversing* the files so that block-sorting worked on preceding contexts. The reversed files gave poorer compression! An example is given in the first columns of Table 1 below. English text files seem to deteriorate slightly with reversal, binary files are neutral, or improve slightly, and program files are also neutral.

Furthermore, reasoning that if block-sorting gave better results with following contexts, then PPM should improve with file reversal, he tried PPM on the reversed files. He found that PPM compressors (and LZ compressors as well) also gave poorer results on the reversed files!

The rest of Table 1 shows tests with Nelson's COMP-2 compressor, running at order 3, and then with LZB. The differences between forward and reversed files are present, but not necessarily as marked as in block sorting, and the overall compression is still worse for both compressors with reversed files. There is no consistency in the results and no real inferences can be drawn. Perhaps the variations just show the reliability of the final compression value and reflect the constancy or otherwise of the statistics over the file. The matter is left as a problem for somebody else to consider.

	BlockSort order-0	Blocksort reverse	COMP-2	COMP-2 reverse	LZB	LZB reverse
bib	2.132	2.182	2.146	2.154	3.174	3.150
book1	2.524	2.549	2.470	2.466	3.861	3.860
book2	2.198	2.218	2.280	2.283	3.282	3.276
geo	4.810	4.748	5.046	5.063	6.173	6.094
news	2.678	2.695	2.716	2.717	3.555	3.564
obj1	4.202	4.201	4.025	3.969	4.266	4.312
obj2	2.709	2.730	2.770	2.802	3.138	3.187
paper1	2.606	2.629	2.542	2.551	2.232	3.230
paper2	2.570	2.593	2.477	2.484	3.434	3.406
pic	0.830	0.830	0.865	0.869	1.030	1.028
progc	2.680	2.683	2.605	2.604	3.082	3.090
progl	1.854	1.851	1.938	1.943	2.117	2.109
progp	1.837	1.838	1.890	1.885	2.086	2.095
trans	1.613	1.625	1.795	1.836	2.124	2.130
Average	2.517	2.527	2.540	2.545	3.111	3.181

Table 1. Compressing with normal and reversed files

Q *The block-sorting algorithm is fundamentally off-line, at least on a per-block basis. Thus it is not attractive for situations (e.g. modems) where latency is important, unless the block sizes are very small.*

A Agreed. It is not really suitable for on-line or serial compression. Wheeler has told me that there is an on-line version of the algorithm, but that decoding is very difficult. File compression and serial, online, compression may well be quite different areas.

Q *I'm not convinced, or at least I'm confused, by your argument in section 5 "The limits to compression." It seems to me that the only coding-length benefit you'll see from a prediction/correction compressor that lacks feedback from the receiver (which you don't have) is the same benefit you get from an arithmetic coder: you don't have to "finish" sending one symbol on a bit-boundary. As I say, perhaps I'm just confused: I think it would help me to understand this section if you drew out your example of the binary symmetric channel with 5/6 error probability. Are you saying that a coder with feedback detection can achieve the limiting channel* *capacity of 0.35? Or does this limit apply only to the forward-correction coder? And why isn't a block-sorting coder just a clever forward-correcting coder, since it doesn't get any feedback from its receiver? (I suspect the problem is that the "errors" in the block-sorting context are not stochastic, so they don't really cut the channel capacity...am I right to suspect that forward-correction is "merely" a way to think about designing compression codes, but there's no theoretical rigour to this construct...yet...maybe this is well-worth thinking through...)*

A Well, yes, it did seem a good argument at the time, but I was starting to have doubts not long after the Report was completed. However, it does indicate a style of compressor different from what is usually considered, and that may be the main benefit.

Q *Your TR doesn't explain how/why the "BS original" coder does a better job on the Calgary Corpus than your most refined coder. (I know no more about the "BS original" algorithm than what you wrote in your TR: that it uses only Huffman and perhaps run length encoding on its output.) Surely an arithmetic coder with a static model can do better than a static Huffman coder, and similarly for the dynamic case: whatever adaptation the Huffman coder is doing, you can mimic, then gain at least incremental coding efficiency by using arithmetic instead of Huffman codes.*

and *I've read your draft paper on block sorting, and am curious why the performance of your order-0 MTF BS does worse than the Huffman MTF BS of burrows & wheeler - it seems to me that using an arithmetic encoder instead of a Huffman encoder should improve performance, not hurt it!!! What do you attribute the change to?*

A I think the problem is one of speed of adaptation, because Burrows and Wheeler did recompute their Huffman code every few thousand bytes. The arithmetic coder adapts quite slowly, especially given that most codes have probabilities well under 1% (see Fig 1 later) and occur only a few hundred times within a 50 kbyte file. (Remember that we are working with MTF codes rather than the original file symbols.)

In any case, it is a widespread misunderstanding that arithmetic coding is somehow, perhaps magically, better than Huffman. For a static distribution for which Huffman coding is optimal, it gives an equivalent result. In general Huffman coding suffers because each codeword must be extended up to an integral bit length, with an average penalty of 0.5 bit/symbol. If we Huffman code the N-th source extension that penalty may be amortised over N symbols and the efficiency improves. Unfortunately coding for even the second extension of a 256 symbol alphabet is hardly practical. Arithmetic coding effectively allows us to code an unlimited extension with reasonable efficiency, and that is the only real difference.

Q *Don't you think the Calgary Corpus is a bit too small to support so much experimentation? I worry that we're* *tuning codes for a very limited universe — over tuning is quite possible. I'd feel more confidence in your experimental results if you tuned your algorithm on one corpus, and tested it on another. The same criticism applies, of course, to nearly every paper (including mine) on data compression...*

and *Maybe it's time to propose a second "Test Corpus" to supplement the Calgary Corpus. The new corpus should have some very long files (tens, if not hundreds, of MB) for applications in high-speed networking. It should have a collection of "random disk-pages" from the disk-traffic of modern PCs and workstations, for applications in compressed DRAM managers. What else?*

A There *is* definitely dissatisfaction with the Calgary Corpus. On the one hand its files are too small – there is increasing demand for compression of files of 100s or even 1000s of megabytes. On the other hand real time disk compression, accesses to compressed data bases and so on use very small units, perhaps 1 kbyte or even less, so its files are also too large! So a new Corpus, or a selection of same, should be certainly in order. I know of several people who are working on the problem. What is really happening is that lossless data compression is becoming an accepted technique in production data processing and is being applied to many many different types of data. Each presumably requires its own appropriate corpus, so that compressors may be compared.

However, the Calgary Corpus exists, and has been used to test a great many compressors. Tests from it can be related easily to other compressors. It does have a variety of files and a compressor which gives good results over the whole corpus is likely to give similar results against another comparable collection. My habit has been to tune a compressor on the file PAPER1, and accept its performance on other files. My experience is that that approach is not too bad. Indeed, some preprocessing such as the run-encoding described later tends to absorb differences so that files are not too different when they actually get to the compressor! I have never managed to make a compressor tune itself dynamically to a file.

Q *The length of the block is "obviously" important. It would be interesting to know how the coding efficiency changes as a function of blocksize. Of course, test file selection is very important for such experiments...*

A This was done by Burrows and Wheeler, for the file BOOK1 of the Calgary Corpus, and for the Hector Corpus, which is about 100 Mbyte of modern English text. Their results are reproduced in Table 2 below. The correspondence between BOOK1 and the Hector Corpus is very close and there is a continued improvement as the blocksize (ie context size) increases.

file	1k	4k	16k	64k	256k	750k	1M	4M	16M	64M	103M
book1	4.34	3.86	3.43	3.00	2.68	2.49					
Hector	4.35	3.83	3.39	2.98	2.65		2.43	2.26	2.13	2.04	2.01

Table 2. Compression (bit/byte) v blocksize for "book1" and the Hector Corpus

Q *Regarding your block sorting compressor: have you computed the entropy of your MTF outputs?*

A This has been done, and is described later in this report. [The suspicion was that the MTF output entropy was quite low and that the coder was so poorly matched to the MTF output that it was increasing the bits/symbol! In fact the measurements show the opposite — the coders code at *better than* the MTF entropy.]

3. Sort improvements.

Two significant changes were attempted in the sort routines, both of which were mentioned in Report 111. The first was the inclusion of a word-oriented sort, as described by Burrows and Wheeler, which can compare 4 or even 8 bytes in a single operation. The second is the use of run-encoding to accelerate the sorting of PIC and similar files.

Word sorting uses an array of long-words to hold the input text, one word per symbol, and "stripes" bytes across preceding words. With 4 bytes per word, a symbol goes into the leftmost byte of "its" word, the second byte of the preceding word, and so on for the two previous words. It is then possible for a word compare to compare 4 bytes at a time, with about the same overhead as for a single character compare. (The comparison uses a stride of 4 words between steps. A 64-bit word can hold 8 bytes and has a stride of 8.) In the actual sort routine the main comparison loop is preceded by a single long-word compare as a preliminary "short compare" filter. The tests can be offset by two positions from the start of the nominal comparands (the first two symbols are known to be the same, because they are in the same radix-sort bucket). The initial comparison then tests the first 6 symbols, or 10 symbols with 64-bit long words.

Run encoding is triggered by detecting a sequence of 4 identical input symbols, and immediately emitting a count of the number of following similar symbols. The symbols for the count are subjected to sorting in the usual way and must be compatible with the rest of the file. In particular, a useful improvement was found from using a 128 symbol alphabet where possible. We therefore split the count into 6-bit groups, transmitted most-significant first. All but the last group have the weight-64 bit set as a flag to indicate that there is more to come. A run with a true length of 4 is then followed by a 0 count and expands to 5 bytes, a run of length 5 has count of 1 and retains its original length, and all longer runs compress.

Run encoding is intended to improve the sorting performance on files like PIC and in this regard is very successful. Sorting PIC on the Macintosh 540C takes about 9 minutes with simple sorting but only 12 seconds with run compression. (About 80% of the digraphs were originally placed in a single sort bucket, largely cancelling any benefits of the radix sort.) With run compression about 75% of the file is absorbed in the runs and it changes to a tractable well-behaved file as far as general compression is concerned. Run encoding also tends to absorb the contexts on which the compression depends and replace them by the unrelated symbols of the run counts. The result is that most files expand by about 0.1% with run compression. PIC improves by about 10%, from 0.93 to 0.82 bits/byte, largely because about 80% of it first disappears in the run compression.

In retrospect, word sorting is of little real benefit. Its main benefit is in accelerating very long comparisons, but most of those very long comparands came from runs which are eliminated by the run encoding. The average number of comparisons varies from about 1.5 words for text files, up to about 7 for the more compressible files like PROGP and TRANS. The time to compare the corresponding 6 to 30 characters is a relatively small component of the total time.

It is worth noting that Burrows [3] has abandoned word sorting in favour of other sorting techniques which are mentioned in their original report and are much more economical in terms of space and time. Word sorting has been retained here simply because it seems quite adequate. For most files the sorting time is about the same as the output encoding time, so there seems little need to alter a generally balanced system.

	MTF entropy	Order-0 new routines	Order-0 "like CACM"
bib	2.286	2.260	2.132
book1	2.759	2.629	2.524
book2	2.397	2.291	2.198
geo	5.336	4.875	4.810
news	2.797	2.733	2.678
obj1	4.105	4.164	4.202
obj2	2.756	2.725	2.709
paper1	2.690	2.693	2.606
paper2	2.704	2.673	2.570
pic	0.876	0.863	0.830
progc	2.705	2.740	2.680
progl	1.903	1.938	1.854
progp	1.871	1.898	1.837
trans	1.633	1.667	1.613
Average	2.630	2.582	2.517

Table 3. Compression with new arithmetic coding routines, unmodified and modified

Another change was to replace the older "CACM" arithmetic coding routines by recently announced equivalents

[7] (which will be known as the "new" routines). The immediate effect was a significant compression degradation! The reasons will be given later, but lie in the management of the associated statistical models. The assumptions underlying the new routines are quite inappropriate to a block sorting compressor. Changing the statistics management to be equivalent to that of the CACM routines largely cured the problem.

The effect is illustrated in Table 3, using an order-0 compressor based on the "new" routines, first unmodified and then modified to act like the CACM routines. An important point is that when these results were taken, the "standard" front end included run-compression, even if only to make it reasonable to compress PIC. The normal compression results relate output bits to input bytes and the MTF_entropy is likewise scaled to reflect the input file size.

Some correspondents have also raised questions concerning the entropy of the MTF output, suggesting that the MTF entropy might be less than the final compression result, because of various inefficiencies in the arithmetic coder. These results are also included in Table 3. For most files the MTF entropy is actually the largest value. The exceptions are for the more compressible text files (PROGL, PROGP, TRANS) where the new routines just cannot follow the local statistics, even while accurately following the overall statistics. In all cases though the more adaptive "like CACM" compressors can exploit the locality of the MTF coding and compress to below the entropy, even with an order-0 compressor. To some extent the adaptation gives the effect of a higher-order compressor.

4. The contexts of block sorting.

Although it was stated earlier that the context structure of the sorted input is quite different from that of a normal file, that structure has not been explained. It is now appropriate to do so, in relation to PPM and block-sorting compression. It has been established that block-sorting is a form of context-driven compression, but with like contexts collected together [5]. The characters which occur in those contexts also appear grouped in the output and a suitable final coder is able to take advantage of the locality.

In PPM compression we develop an extensive history of known contexts. Each symbol is predicted from its preceding contexts and then used to extend existing contexts and develop new ones. The multiple contexts develop in parallel and the relevant contexts usually change completely as each symbol is processed. For any particular input symbol the encoder and decoder can develop an appropriate collection of contexts of various orders, predict the probabilities of each possible symbol, and code according to those probabilities.

The accurate prediction of probabilities and especially of the probability of escaping into lower order contexts is the crux of successful PPM compression. Improvements generally follow from improved prediction of these probabilities. Most importantly, with PPM processing the input sequentially we always have precise knowledge of all possible contexts.

In block-sorting compression all similar contexts are collected together in a region of the reordered input. As we proceed we develop one context and, when it is completely processed, move on to another context, possibly similar, but possibly quite different. Thus whereas PPM develops its contexts in parallel, block sorting develops the same contexts sequentially. PPM expects *major* context changes between successive characters, whereas block-sorting expects *minimal* changes – with MTF coding emitting mostly 0s, with a few 1s and possibly 2s. When there is a major context change, such as "az…" to "ba…", block sorting has to readjust and learn new statistics, and readjust quickly.

If we examine the output of a block sorting compressor, it is immediately obvious that there is little relation between the output symbols and changes in their contexts. There may be frequent changes of symbol related to a relatively high order context change — as an extreme example if the context is the very frequent " the ", the emitted symbol is largely determined by changes in the 6th and higher orders. Few PPM compressors look that far away. Again, a change of order-1 context (perhaps "by…" to "ca…") may be marked by a simple change from one run to another.

So there is no way to know that the block-sorting environment has changed. The coder knows, but that is privileged information which is available to the decoder only after the whole file is reconstructed. (In contrast the coder and decoder in PPM have exactly the same information and both know all context changes.) We must accept that block-sorting compression, while it exploits contexts of unlimited length, destroys those contexts in the sorting and makes them completely invisible. All that the coder and decoder can do is to respond to local changes in what seem to be the likely symbols.

We can now comment on the relation of the new arithmetic coding routines to the older "CACM" routines and the effect on compression. The CACM routines had a frequency limit which was small compared with most file sizes, forcing frequent rescaling (halving all counts) as a file was processed. The frequency increment remained constant, so that older symbols tended to have less influence and older contexts were gradually forgotten. The new routines however allow very large counts. When rescaling is necessary, the counts and the increment are both scaled so that their relative magnitudes remain the same. Thus symbols have uniform treatment across the whole file; old counts are just as important as recent ones and older contexts are not forgotten.

The behaviour of the new routines is quite appropriate for PPM compressors where multiple contexts are developed incrementally and the history across the whole input file is relevant. The behaviour is decidedly inappropriate for block sorting where contexts are ephemeral and may have to be forgotten quite quickly. This is why the new routines perform so poorly on block sorting and had to be revised in line with the CACM behaviour.

5. Hierarchical coding models.

The remainder of this report deals with improvements to the initial Order-0 compressor, and is presented in largely chronological order. This approach may give some feel for the understanding which gradually developed.

From earlier experiments, reported in Report 111, it was clear that the sorted output was handled poorly by conventional "good" compressors because of the total rearrangement of the context structure. The historical context on which they relied became essentially a local context which the compressors could not handle. The questions were "What was the actual structure of the rearranged text?", and "What was the best coder to exploit this structure?" In all cases the coding was done after the Move-To-Front processing, so we were dealing with the very skew symbol distributions which were noted in Report 111. Figure 1 reproduces some of those distributions.

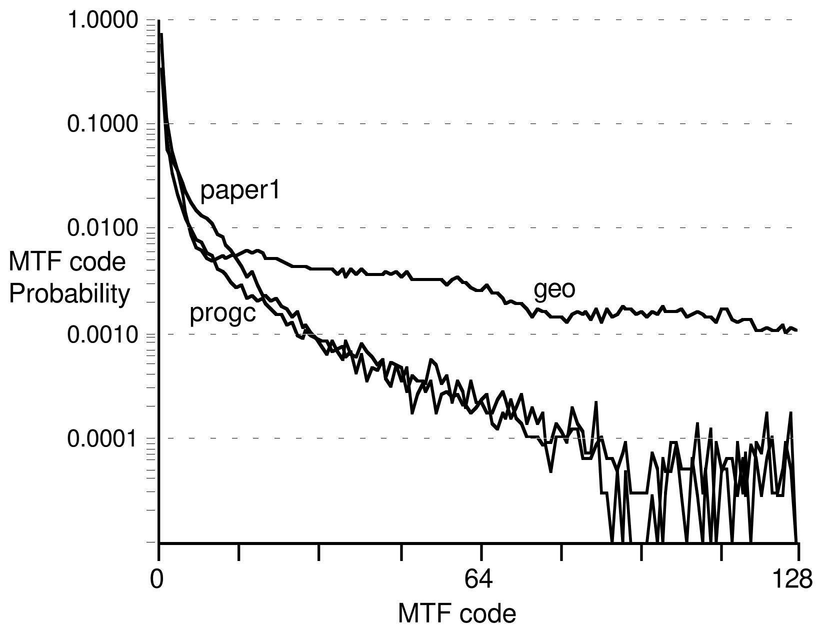

Figure 1. MTF code probabilities for three files

There are two areas to these distributions, areas which are somewhat concealed by the logarithmic scales. The first half-dozen or so most frequent symbols tend to be quite active (the actual symbols of course changing throughout the file), with the remaining symbols being much like a noisy background. One difficulty, which will be addressed later, is that the MTF action disguises the individual symbols and we can take little advantage of any inherent frequency differences except that frequent symbols will tend to be closer to the head of the MTF list.

The distributions shown are large-scale ones covering the whole of the file. In the active area of the distribution we can expect to see significant local changes throughout the sorted file. In some cases there will be several active symbols, all with high frequencies, while other cases will be dominated by long runs of single symbols. The long term average distribution is a compromise which does not represent many of these extremes particularly well and we may expect better compression from models which can represent local features. Adjusting to contexts which may change within a few tens of symbols requires arithmetic coding models with limits of a few tens of counts and handling only a few symbols. A complete model with all the symbols and thousands of counts simply cannot adjust within the necessary time scale.

The "order-0" statistical model of the MTF output is at best only an approximation or averaging-out of the local contexts, probably with considerable local deviation from that model. Figure 2 shows the frequencies of the first few MTF ranks for four successive samples of 100 symbols after MTF coding the file PAPER1. It is obvious that there are enormous differences between adjacent samples; even in these few samples, the counts vary from 30 – 65 for 0, 3 – 7 for 2 and 1 – 9 for 4, showing the enormous local variations from the overall "average distribution". There is little correlation between the frequencies of differing ranks. It is never greater than 65% and usually less than 20%. The differences are a natural consequence of having quite unrelated contexts following one another in quick succession, each with its own "signature" or combination of MTF probabilities.

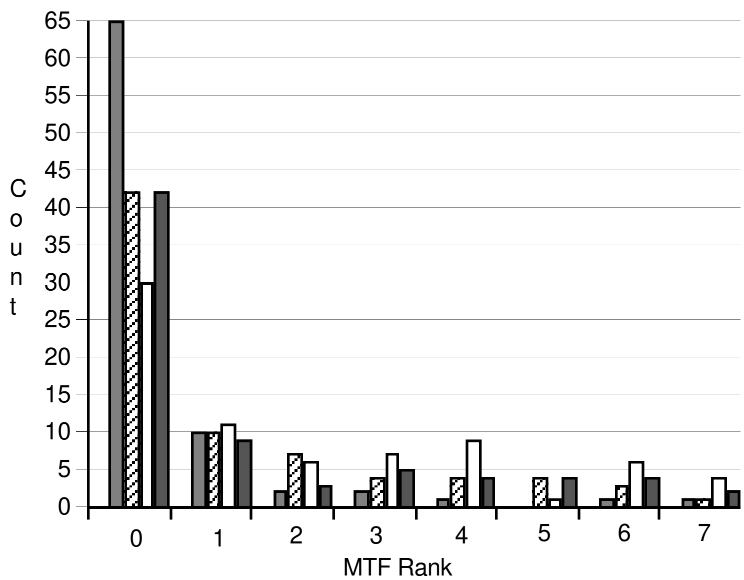

Figure 2. MTF code frequencies for successive areas of a sorted file

Improving compression over that achieved by the order-0 model requires models which can adapt quickly to local changes in frequency, especially for the first half dozen or so MTF codes. Early work therefore involved the use of a "cache" to give preferential treatment to the much more likely low-valued MTF codes, working with a small alphabet and adjusting to local changes much more quickly than could a full-alphabet encoder. Using a two-level coding model, one for the more frequent symbols and allowing escapes to the full model for less frequent symbols, gives a definite improvement in compression. This contradicts the conventional lore that an arithmetic coder adjusts optimally to any symbol distribution, but that wisdom takes little account of the fine structure which is so important here.

An early clue as to the validity of the cache coding model came in experiments with an order-1 compressor which was found to have a bug which prevented it from emitting from other than a preceding 0-value context. It would emit runs of zeros with no problem, but always escape to order 0 for any other symbol. Fixing the problem made the compression worse!

The other possibility was run-encoding the MTF output, which was apparently done by Burrows and Wheeler. We need a very efficient run-length coding to compare with the cache-based methods. The cache models can encode 0s at about 0.2 – 0.5 bit/byte for most files, so even a 1-byte length field corresponds to 16 to 40 coded values. (The situation is related to the efficient coding of the MTF positions. In both cases the frequency distributions are highly skewed and are most efficiently coded by a unary encoding, at least for small values; the large values are so rare that their precise coding is relatively unimportant anyway. There is therefore no benefit from using a coded value for the run-length. With Huffman coding, which cannot represent fractional bits, there may be advantages from run coding.)

	Run length code + Cache	2-level cache 4&16	BW94	PPMC
Bib	2.024	2.035	2.070	2.110
Book1	2.460	2.450	2.490	2.480
Book2	2.116	2.109	2.130	2.260
Geo	4.573	4.622	4.450	4.780
News	2.564	2.566	2.590	2.650
Obj1	4.007	4.029	3.980	3.760
Obj2	2.552	2.560	2.640	2.690
Paper1	2.531	2.516	2.550	2.480
Paper2	2.492	2.486	2.510	2.450
Pic	0.835	0.857	0.830	1.090
ProgC	2.572	2.562	2.580	2.490
ProgL	1.784	1.770	1.800	1.900
ProgP	1.762	1.747	1.790	1.840
Trans	1.541	1.519	1.570	1.770
AVG	2.415	2.416	2.427	2.482

Table 4. Results with cache-organised coding models

After numerous experiments, two generally competitive coders emerged.

1. A strictly hierarchical model, with caches of 4 and 16 entries in front of the full order-0 model. If the code cannot be emitted from the first-level cache (4 entries — 0–3), an escape code signals use of the second-level cache (values 4 – 15), and that can in turn escape to the order-0 model (values 16 – last). "Fetching" into the caches is left to the MTF coder, as is aging of unused entries.

2. A hierarchy of run-length coding for zeros, an 8 entry cache for the more frequent codes, and then the full

order-0 model. The run-coding uses a simple unary code, emitting a 0 for a zero and then a 1 as a final escape code to signal the end of the run. With the average run being about 6 symbols, this is more efficient than emitting an explicit count. An encoded run is triggered by a preceding 4 identical symbols.

Escapes are simply treated as another symbol and allowed to find their own natural probability. A better method of calculating escape probabilities would undoubtedly improve compression but, as explained earlier, we have very little knowledge of the specific contexts and therefore no obvious way of calculating escape probabilities from those contexts. The results are shown Table 4, together with the original block-sorting compressor (BW94), and PPMC as a reference compressor. Even with the allegedly more efficient arithmetic coding, neither of the new techniques is much different from the BW94 version, although all are rather better than PPMC on most files.

6. Delta coding models.

Report 111 discussed some of the very early work by Shannon on the compressibility of English text [8], and showed how his coding model was related to the MTF-coded output of a block-sorting compressor. Briefly, a coder (human or inhuman) estimates the most likely symbol in some situation. An associate who (which) can see the text tells if the estimate is correct. If wrong, the next most likely symbol is suggested – it may be accepted or rejected, and so on until the estimate is correct. The "emitted" code is then a sequence of "yes" and "no" symbols (or 1s and 0s), where a 1 marks a correct prediction and a 0 an incorrect prediction. A decoder with a matching coding model can use the sequence of emitted 0s and 1s to recover the original text. The essence is that the coder predicts a list of symbols in probability order and steps along that list. With MTF coding, we have a unary coding of the MTF distances.

Using a single binary model for the 0s and 1s gives a moderate compressor (2.67, see Table 4 of Report 111). Using two models (the "Delta2" coder), one mostly 1s and one mostly 0s, gives useful improvement as shown in Table 4 below. (The values are different from those in Report 111 because of the new arithmetic coding routines and because we now use a run-encoding preprocessor.) This table includes results from the recently published PPM* compressor [5] as recent improvement on PPMC. From here we take two different routes.

The "Delta2" coder has two coding models. The "zero" model is used to emit MTF codes of 0, and the first 1 of the unary code of any non-zero code. The "one" model emits the unary codes for the other symbols and the first 0 code of a run of 0s. (The model is chosen for a symbol depending on whether the previous symbol is zero, and within a symbol the "one" model is forced as soon as a 1 bit is emitted.) To force a rapid adjustment of the models to changing contexts, we introduce a measure of amnesia into the coding models by halving the counts of the zero model whenever a non-zero symbol is emitted and halving the counts of the one model whenever a zero symbol is emitted. The effect is that "1s" tend to be forgotten during the emission of a sequence of 0s, and that 0s are forgotten while emitting 1s. The result is the "Delta2 - halve" model of Table 5, with a performance which is clearly comparable to PPMC and BW94, and quite surprising for such a simple model. The improvement is especially significant for the less-compressible binary files GEO and OBJ1.

	Delta2	Delta2 - halve	Delta3	DeltaN	PPMC	PPM*
Bib	2.036	2.020	2.043	1.976	2.110	1.910
Book1	2.449	2.454	2.442	2.399	2.480	2.400
Book2	2.104	2.098	2.101	2.056	2.260	2.020
Geo	5.436	4.541	4.488	4.547	4.780	4.830
News	2.640	2.620	2.607	2.526	2.650	2.420
Obj1	4.504	4.163	4.022	3.928	3.760	4.000
Obj2	2.817	2.699	2.585	2.488	2.690	2.430
Paper1	2.576	2.561	2.566	2.480	2.480	2.370
Paper2	2.490	2.484	2.495	2.438	2.450	2.360
Pic	0.802	0.787	0.789	0.774	1.090	0.850
ProgC	2.656	2.626	2.614	2.517	2.490	2.400
ProgL	1.815	1.812	1.812	1.734	1.900	1.670
ProgP	1.845	1.827	1.813	1.720	1.840	1.620
Trans	1.647	1.643	1.623	1.502	1.770	1.450
AVG	2.558	2.452	2.429	2.363	2.482	2.338

Table 5. Results with delta coding models

A parallel approach which yields very similar results comes from considering what should be done when a new symbol is fetched (MTF index > 0). Are we to assume that the new symbol will continue, or will we fetch another symbol, possibly even the one just left? Without any better predictor, we assume that the two possibilities are equally likely and reset the zero model to equiprobable symbols. This gives a modest improvement on some files (< 0.3%) but overall is about equivalent to halving the model.

7. Extended delta coding.

The other route to delta coding is that indicated in Report 111. It can be very inefficient emitting a large-valued unary code, so after some threshold, we just emit the actual value. If, for example, the threshold is 4, a value greater than 4 will be emitted as "11111*xxxx*…". The result is the Delta3 entry of Table 5 ("Delta coding with 3 models"). It is again comparable to the "Delta2 – halve" and the BW94 coders.

There are two possible views of this coding —

1. As described in Report 111, the coding approach is justified by the distribution of the MTF codes. If we have a very skew distribution, with successive symbol probabilities falling off by more than a factor of 2, an optimum code generates a sequence of unary codes in that region. If we look at the distributions in Fig 1

earlier, we see the sharp fall followed by a slower fall, the two corresponding approximately to the initial unary coding and later block coding.

2. In more human terms and perhaps reflecting Shannon's approach, after some number of wrong guesses the frustrated estimator is allowed to ask for the actual code, at a penalty of say 6 or 8 bits.

As a final improvement at this stage, we introduce separate coding models for each of the digits of the unary coding of the MTF values. The skewness of the distribution of MTF values means that the relative frequencies of 0s and 1s are quite different across the different digits. The result is the DeltaN line in Table 5, with a performance quite comparable to that of PPM*. (There is no improvement from using different models for successive 0s. There the symbols are essentially equivalent, with each having the same probability of ending or continuing a run.)

It is interesting that two approaches have converged to quite similar models. Both use a special binary model for 0-symbols and a conventional large-alphabet model for lower-frequency codes. The differences lie in the handling of MTF codes in the range from 1 to about 5. In one case we use a small "cache" to hold all of the codes, with escape symbols to move between the different levels. In the other case we use unary coding to cover this range, with a range of binary models. The cache-based coders suffer slightly because their escape symbols are not truly integrated into the coding process and always appear as an extraneous symbol, whereas the delta coders can always use a 0 or 1 as part of the next symbol. In both methods the "small-value" coding models have small count limits and relatively large increments to force rapid adaptation to the locally-changing symbol frequencies.

"DeltaN", the best of all the compressors developed here, probably wins because of its absence of escape coding. The other good compressors all use some form of escape from runs or from caches. Although, as shown in Report 111, an escape code is properly regarded as a prefix to the following code, that assumes that we know the correct escape probability. In no case here do we know that probability and escape coding may be a significant source of inefficiency.

8. LZ-77 preprocessing.

Report 111 mentioned the possibility of using LZ-77 preprocessing for accelerating the sorting of files with extensive repeated or cyclic sequences. LZ-77 preprocessing seemed an interesting possibility anyway because of previous work by Peter Gutmann on combining LZ-77 and arithmetic coding in his LZA*n* series of compressors.

A simple LZ-77 compressor was therefore coded as a preprocessor to the main block-sorting compressor. Because all of the LZ output is subjected to the sort its output must be compatible with the normal file symbols. LZ-77 phrases are therefore represented by a sequence *Ennnddd*, where *E* is an escape code (such as the ASCII ESC, 0x1B), *nnn* is 1–3 length symbols and *ddd* is 1–3 displacement symbols, Both *nnn* and *ddd* use the code described earlier for run-lengths, with 6 information bits and a flag to indicate continuation to another symbol.

Table 6 shows some results for the smaller files of the Calgary Corpus. There is no benefit at all for most files in having LZ-77 preprocessing. In fact the compression is usually worse for *any* LZ-77 minimum phrase length or combination of LZ-77 parameters, even though the LZ-77 step often reduces the file size to 35–50% of the original size. (The compression here is modest because the LZ-77 step is not tuned for good compression.) Both block sorting and LZ-77 compression rely on the longer phrases; the LZ-77 processing turns these compressible phrases (or contexts) into sequences of a few bytes which are essentially random and quite incompressible.

	Order-0	Order-0 LZ	DeltaN	DeltaN LZ
Bib	2.132	2.597	1.976	2.214
Geo	4.810	4.774	4.547	4.548
Obj1	4.202	4.126	3.928	3.955
Paper1	2.606	2.897	2.480	2.587
Paper2	2.570	2.867	2.438	2.560
Pic	0.830	0.830	0.774	0.807
ProgC	2.680	2.717	2.517	2.616
ProgL	1.854	1.887	1.734	1.810
ProgP	1.837	1.846	1.720	1.793
Trans	1.613	1.626	1.502	1.637
AVG	2.513	2.617	2.362	2.453

Table 6. LZ-77 preprocessing; shortest LZ-77 phrase = 15

LZ-77 preprocessing is not therefore a reasonable option for improving the overall compression, but that was not really the intention. It is good for its intended purpose of accelerating the sorting of some pathological cases. As an extreme case, a file containing 2000 repetitions of the sequence "*aaaaaaaaab*", to a total length of 20,000 bytes, which took about 15 minutes to sort with older sort routines, completed processing in under 1.5 seconds with LZ preprocessing. (The compression was to 16 bytes, 0.0064 bit/byte, which while not as good as has been reported for files of this type, is not too bad, considering the relatively poor LZ phase. A 100 kbyte extended version of the file also produced 16 bytes of output and ran in 3.5 seconds.)

Both run-encoding and LZ-77 encoding are satisfactory solutions to the problem of the pathological input files which sort very slowly. Run encoding can be included without too much penalty, but LZ-77 encoding clearly needs a trial scan of the input, accepting its output only if the file reduces to about 10% of the original size from the LZ-77 compression.

9. Direct coding.

Report 111 mentioned the possibility of using direct encoding of the sorted output, without the intermediate MTF stage. The hope was that there might be some gain from working with explicit characters, whereas MTF coding essentially hides the identity of characters, replacing all by a simple recency index. A simple direct encoding of the output gives the results in the first column of Table 7, with a frequency limit of 10,000 and an increment of 600. The relatively large increment forces a rapid adaptation of the model to local statistics. If the limit is too low, the model cannot accommodate differences in the less-frequent symbols, while if it is too high the less-frequent symbols may be too expensive to encode; the value is not however as critical as that for the increment (or more correctly the ratio of limit : increment).

To improve the coding we introduce (again!) a cache for the more frequent symbols. It is operated with a relatively small limit and large increment to force adaptation. It is a "full-alphabet" model able to hold any combination of source symbols (but only a few of them), but has the unique feature that it is designed to forget infrequent symbols. In normal arithmetic coding we round upwards when halving symbol frequencies so that frequencies never drop below 1. Here we halve with truncation and allow infrequent symbols to go to a frequency of 0 and disappear from the cache.

This change required the development of a "sparse" model for the new coding routines. It uses a table of the current symbols and their frequencies (up to about 16 in all) with simple serial scans through the table for searching, updating, etc. For small alphabets this is as fast as the Binary Indexed Tree used in the main routines. For sparse alphabets it is much faster because we act on only the active symbols.

Thus we initially attempt to emit the symbol from the cache; if that fails we emit an Escape from the cache model to emit from the main model. The emitted symbol is then brought into the cache. In line with earlier observations, we halve the cache at each miss so that older symbols are forgotten. Results for this model are in the second results column of Table 7. There is some improvement over the simple model, but it is still not as good as even an order-0 coding with MTF.

At least some of the deficiencies lie in the handling of frequencies in the cache and are very similar to the problems of encoding from within contexts in PPM.

1. **Escape frequency.** PPM estimates escape probabilities (the probabilities of encountering unexpected symbols) from the number of times that symbols have been seen in the current context. Something in the same spirit is needed here, but we have no knowledge of the precise context and that makes prediction very difficult.

2. **Initial Frequency.** Irrespective of how it is handled later, a new symbol must appear with a high initial frequency because it may well be the first symbol of a run. However it may be followed by another new symbol, or a reversion to the previous symbol. In the circumstances we can only make a reasonable guess. A new symbol is therefore forced to the same frequency as the last symbol processed, making the two equiprobable.

	Direct-0	Direct - cache	direct cache2	BW94	PPMC
Bib	2.353	2.236	2.188	2.070	2.110
Book1	2.536	2.617	2.592	2.490	2.480
Book2	2.259	2.235	2.213	2.130	2.260
Geo	5.646	5.064	4.632	4.450	4.780
News	2.896	2.755	2.719	2.590	2.650
Obj1	4.581	4.520	3.982	3.980	3.760
Obj2	2.985	2.712	2.647	2.640	2.690
Paper1	2.901	2.888	2.690	2.550	2.480
Paper2	2.751	2.780	2.630	2.510	2.450
Pic	0.860	0.839	0.807	0.830	1.090
ProgC	2.987	2.942	2.689	2.580	2.490
ProgL	2.189	2.063	1.919	1.800	1.900
ProgP	2.168	2.068	1.896	1.790	1.840
Trans	2.061	1.862	1.751	1.570	1.770
AVG	2.798	2.684	2.525	2.427	2.482

Table 7. Results with direct coding models

Some initial attempts were made to tune the model in line with the above principles, giving the results shown in the "direct cache – 2" results of Table 7. The escape frequency calculation is similar to "method A" of PPM, with a running count incremented by 1 for a cache hit and halved with upward rounding for a cache miss. The escape frequency is then equal to the total frequency divided by the current count. The initial symbol frequency is set as described.

Even with these improvements, the results are not good because of the difficulty determining the contexts. It does not seem possible even to determine a change of order 1 context with any reliability. Thus a symbol in the cache may belong to the current context, or it may be an old, irrelevant, one which is lingering around and really should be forgotten (and is interfering with compression).

10. Direct coding with augmented alphabet.

While the "direct coding" methods work, they do not work that well. Part of the problem is that simple models just do not adapt well to changing contexts — it is difficult for a new symbol to become dominant after a short run, or for an older symbol to lose dominance. These problems can be reduced by introducing an extra symbol which simply denotes a repetition of the previous symbol. There is an obvious relation to the encoding of runs, or to the dominance of code-0 in MTF coding. Here the symbol is an extension to the normal alphabet. When a symbol is first referenced in a run it is fetched from the normal model, but thereafter it is handled by the added "Repeat" code.

The improvement over the simple direct coding is about 11% and brings the overall performance to almost exactly that of the corresponding Order-0 compressor with MTF. This is a somewhat pleasing result as it shows that the MTF step is not always necessary and can be omitted with an appropriate coding model.

Attempts to improve on this result were markedly unsuccessful. For example, combining the augmented alphabet with the fast-adjusting cache gave the results in the second to last column of Table 8. While some binary files are improved, most files are worse and the overall performance is reduced.

	order-0 MTF	Direct - simple	Direct - "Repeat"	Hybrid MTF—Dir	Dir-cache "repeat"	Context cache
bib	2.132	2.353	2.126	2.126	2.163	2.342
book1	2.524	2.536	2.553	2.568	2.601	2.738
book2	2.198	2.259	2.211	2.211	2.246	2.371
geo	4.810	5.646	4.814	4.842	4.695	5.002
news	2.678	2.896	2.643	2.658	2.676	2.867
obj1	4.202	4.581	4.164	4.136	4.070	4.122
obj2	2.709	2.985	2.664	2.666	2.641	2.847
paper1	2.606	2.901	2.625	2.608	2.668	2.813
paper2	2.570	2.751	2.588	2.586	2.627	2.728
pic	0.830	0.860	0.791	0.803	0.806	0.826
progc	2.680	2.987	2.686	2.662	2.707	2.845
progl	1.854	2.189	1.853	1.850	1.877	2.065
progp	1.837	2.168	1.851	1.832	1.855	2.068
trans	1.613	2.061	1.606	1.601	1.621	1.914
Average	2.517	2.798	2.513	2.511	2.518	2.682

Table 8. Compression with augmented alphabet

Another approach (from an idea of Charles Bloom [2]) involved a "context cache" based on the last few emitted symbols — it has obvious similarities to PPM coding on the sorted output. That too did not work well, with results very similar to those with a simple cache. The problem here is that it is very difficult to switch quickly from one high-probability symbol to another high probability symbol. The older probability must decay as the new one grows and there may be little stability before the context changes yet again. Actually, a hint that this approach might not work is found in the earlier experience with context-compressing the sorted output. Most of those compressors used orders of 3 – 6, which is comparable to the best size of the context cache here. The situation would certainly be different if we could determine the contexts and therefore the conditioning classes for the symbols, but that does not seem possible.

Using the "repeat last symbol" code is actually identical to having a special symbol whenever the MTF position is zero. We can augment the alphabet further with special symbols for the first few (most probable) MTF codes, emitting the MTF codes where appropriate and normal symbols otherwise. It is effectively a "hybrid" between

MTF and direct coding. The results are shown in the “hybrid” column of Table 8, with special codes for MTF positions of 0 and 1. There was no benefit in coding larger MTF positions, and even here the overall results are essentially the same as for the “repeat last symbol” coding.

Thus, direct-encoding of the sorted output with a very slightly extended order-0 model gives very similar results to encoding the MTF output with an order-0 model, but there seems to be no more possible improvement.

11. Processing resources.

Compressors can be compared on the basis of speed as well as compression achieved. Here the speeds are compared for the block-sorting Order-0 compressor, Nelson’s COMP-2 order=3, and Bell’s LZB compressor (8K window, binary tree structure). The programs were run on a Macintosh Powerbook 540C (66MHz 68040LC) and on a HP 755 Workstation (99MHz PA-RISC). The results are in Table 9. Also shown are times for PPMD+[9], representing a current state of the art compressor. Finally, the last column shows the compression times for “bred” a recent version of the BW94 algorithm, running on a DEC5000/133. This uses Huffman coding of the output and Burrows’ fast sorting routines as described earlier

	Macintosh 540C			Hewlett-Packard 755				
	BS-order0	COMP-2	LZB	BS-order0	COMP-2	LZB	PPMD+	bred
bib	16.6	25.0	13.8	5.1	10.2	2.2	21.3	2.50
book1	123.9	166.3	88.8	40.0	75.1	14.2	214.7	27.60
book2	98.9	132.9	73.7	30.7	57.4	12.0	153.4	20.10
geo	15.0	93.6	12.7	4.9	34.9	1.9	279.8	3.90
news	62.7	104.8	45.4	17.9	41.4	7.6	105.9	11.40
obj1	3.3	14.0	5.6	0.9	5.6	1.1	8.0	0.40
obj2	41.3	97.2	36.6	11.5	36.8	6.4	92.6	6.80
paper1	7.4	13.2	7.2	2.2	5.2	1.1	25.2	1.00
paper2	11.3	18.9	10.4	3.6	7.9	1.6	15.6	1.80
pic	23.7	135.0	352.5	7.1	63.6	93.4	249.0	4.50
progc	5.7	10.9	5.3	1.6	4.2	0.8	7.9	0.60
progl	12.6	15.1	11.6	3.1	6.4	2.1	13.0	1.50
progp	12.9	11.4	8.5	2.2	4.7	1.5	10.8	0.90
trans	25.4	20.1	14.6	4.5	8.4	2.8	16.9	1.90
Total	437.0	723.4	334.2	128.2	298.2	55.3	965.1	84.9
Ratio to Block-Sort		60%	131%		43%	232%		

Table 9. Compression times (seconds)

The file PIC is anomalous and shows up particularly well with block sorting because of the run-compressing preprocessor; the long runs slow down all other compressors. The total times therefore exclude the PIC values. On the other files LZB is usually the fastest, with block sorting not much slower but of course giving much better compression. COMP2, with similar compression to clock sorting, is usually somewhat slower. These results differ from those of Burrows and Wheeler because they had gone to much more trouble to optimise the speed of block sorting. Their sorting phase was faster, as was their Huffman final encoder.

Compression times are shown for the simplest, "Order 0" compressor. Most of the better compressors are slightly slower because they may need several stages of arithmetic coding for each symbol. The "Delta 2" with its full unary coding is the worst in this regard, especially on GEO where many symbols have large MTF displacements and may need tens of coding steps. The "Delta N" compressor needs some extra coding steps and is about 5–10% slower than Order 0.

The memory requirements vary widely. LZB, with an 8 Kbyte window, requires only 200 Kbyte of workspace. COMP2 did not compress GEO at all until it had about 5 Mbyte of storage available; its rather complex storage structure makes it difficult to estimate the requirements of other files. Block sorting needs 9 bytes for each data byte (with word-sorting), and another 786 Kbyte for its radix 65,536 sorting tables. Most files of the Corpus can be processed in less than 2.5 Mbyte; BOOK1 needs about 8 Mbyte.

Table 9 also shows times for PPMD+, the best compressor known to me [9]. (Its compression results are reported in the next section.) On a 50MHz SPARC server it requires about 1,200 seconds to compress the Corpus (965 seconds without PIC), with storage of 12 – 20 data bytes per input byte on most files. Table 9 also includes very recent results for "bred" a version of BW94 which has been recently announced by Wheeler [10]. It will be commented upon later.

Thus block sorting is certainly closer in speed to LZ compression, while giving the compression of PPM-style compressors, with memory requirements between the two. This is line with the two reports of Burrows and Wheeler [4, 10].

12. Other recent work.

When this report was almost completed, some more results were released by Wheeler[10], including implementations of routines very similar to the original BW94, and a short report on some of his more recent work. The coding uses a hierarchical coding model, together with run encoding of consecutive MTF 0s. The first level of the hierarchy uses a 2-bit code; successive bits of the run length are coded as 00 for 0s and 01 for 1s, an MTF position of 1 is encoded as 10 and the code 11 is used as an escape to a general model for the other positions. Using 3 of these codes (6 bits) as a context for arithmetic coding gives results which are quite competitive with any other compressor and are shown in Table 10 as "BW95 6/2 arith".

13. Conclusions.

Block sorting is clearly the basis for a family of good compressors. Even though they do not compress quite as well as some recently-presented techniques, the block-sorting compressors are nearly as good, as well as being faster and with reasonable memory requirements.

Table 10 presents the performance of the best compressors of each type tested here, together with results for some other benchmark compressors —

- order-0 MTF block sorting, with order-0 arithmetic compression of the MTF output
- PPMC the established standard for quality compression
- BW94 the original block-sorting compressor, as published by Burrows and Wheeler
- PPM* a recently published unbounded context version of PPM,
- PPMD+ a further-improved version of PPM
- BW95 6/2arith the best of Wheeler's recent compressors

The best of the improved block-sorting compressors in this report is nearly as good as PPM*, but still 3.5% inferior to PPMD+. For practical considerations a 4% difference in compression is of little significance (apart from in text compression contests!) and considerations such as memory requirements and compression speed are likely to be as important. In this regard block-sorting shows up especially well. It is several times faster than PPMD+ and uses about half the memory, or less. (The recent BW95 compares *very* well against PPMD+.)

	order-0 MTF	Direct - "Repeat"	2-level cache	DeltaN	PPMC	BW94	PPM*	PPMD+	BW95 6/2 arith
Bib	2.132	2.126	2.035	1.976	2.110	2.070	1.910	1.862	1.90
Book1	2.524	2.553	2.450	2.399	2.480	2.490	2.400	2.303	2.36
Book2	2.198	2.211	2.109	2.056	2.260	2.130	2.020	1.963	1.98
Geo	4.810	4.814	4.622	4.547	4.780	4.450	4.830	4.733	4.69
News	2.678	2.643	2.566	2.526	2.650	2.590	2.420	2.355	2.50
Obj1	4.202	4.164	4.029	3.928	3.760	3.980	4.000	3.728	3.78
Obj2	2.709	2.664	2.560	2.488	2.690	2.640	2.430	2.378	2.40
Paper1	2.606	2.625	2.516	2.480	2.480	2.550	2.370	2.330	2.38
Paper2	2.570	2.588	2.486	2.438	2.450	2.510	2.360	2.315	2.37
Pic	0.830	0.791	0.857	0.774	1.090	0.830	0.850	0.795	0.76
ProgC	2.680	2.686	2.562	2.517	2.490	2.580	2.400	2.363	2.41
ProgL	1.854	1.853	1.770	1.734	1.900	1.800	1.670	1.677	1.61
ProgP	1.837	1.851	1.747	1.720	1.840	1.790	1.620	1.696	1.61
Trans	1.613	1.606	1.519	1.502	1.770	1.570	1.450	1.467	1.39
AVG	2.517	2.513	2.416	2.363	2.482	2.427	2.338	2.283	2.296
rel to	110.3%	110.0%	105.8%	103.5%	108.7%	106.3%	102.4%		100.5%

Table 10. Summary of results : best compressor of each type

Even though it means that we lose information on the exact symbol being encoded, it is necessary to retain the Move-To-Front step between the sorting and the final coding; direct coding of the sorted output does not give good results.

The main problem with block-sorting compression is that, although the technique uses contexts of unbounded order, it can use no information on those contexts. We do not know when an old context disappears and its symbols should be forgotten, or when we are starting a new context with a new and unique symbol and must clear the context. If these problems of context recognition can be solved, or we can obtain very efficient fast adaptation to local contexts, there could be room for useful improvement over what has been presented here.

Run compression of the raw input, initially used to improve the sorting speed, gives a reasonable improvement on PIC with its many repeated contexts. These repeated contexts also slow down PPM and LZ-77 compressors; it may be interesting to consider the effect of a preliminary run compression on their performance.

14. Acknowledgements.

This work was supported by grant A18/XXXXX/62090/F3414032 from the University of Auckland and performed while the author was on Study Leave at the University of California – Santa Cruz and the University of Wisconsin – Madison. The author acknowledges the contributions of all of these institutions.

Many people gave me useful comments and encouragement following the publication of my earlier Report 111. Some of their comments are answered at the start of the present report. In particular I thank Profs D. Wheeler, R. Brent and C. Thomborson, and Alastair Moffat (who provided the new arithmetic coding routines), Bill Teahan (for the PPM+ results), Mike Burrows and Charles Bloom.

References

[1] T.C. Bell, J. G. Cleary, and I. H. Witten, "*Text Compression*", Prentice Hall, New Jersey, 1990

[2] C. Bloom, private communication

[3] M. Burrows, private communication.

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

[5] J. G. Cleary, W.J. Teahan, I. H. Witten, "Unbounded Length Contexts for PPM", *Data Compression Conference, DCC-95*, Snowbird Utah, March 1995

[6] "Experiments with a Block-Sorting text Compression Algorithm", The University of Auckland, Department of Computer Science, Technical Report 111, March 1995.

`ftp.cs.auckland.ac.nz /out/peter-f/report111.ps`

[7] A. Moffat, R. Neal, I.H. Witten, “Arithmetic Coding Revisited”, *Data Compression Conference, DCC-95*, Snowbird Utah, March 1995

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

[9] W.J. Teahan, private communication

[10] D.J. Wheeler, private communication [This communication was also posted to the `comp.research` newsgroup. The files are available by anonymous FTP from `ftp.cl.cam.ac.uk` in the directory `/users/djw3`]