Improving semistatic compression via pair-based coding*

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[2] and José R. Paramá[1]

[1] Database Lab., Univ. da Coruña, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain. {brisaboa,fari,parama}@udc.es
[2]Dept. of Computer Science, Univ. de Chile, Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

Abstract. In the last years, new semistatic word-based byte-oriented compressors, such as Plain and Tagged Huffman and the Dense Codes, have been used to improve the efficiency of text retrieval systems, while reducing the compressed collections to 30–35% of their original size.
In this paper, we present a new semistatic compressor, called *Pair-Based End-Tagged Dense Code (PETDC)*. PETDC compresses English texts to 27–28%, overcoming the optimal 0-order prefix-free semistatic compressor (Plain Huffman) in more than 3 percentage points. Moreover, PETDC permits also random decompression, and direct searches using fast Boyer-Moore algorithms.
PETDC builds a vocabulary with both words and pairs of words. The basic idea in which PETDC is based is that, since each symbol in the vocabulary is given a codeword, compression is improved by replacing two words of the source text by a unique codeword.

1 Introduction

The grown in size and number of text databases during the last decade makes compression even more attractive. Compression exploits the redundancies in the text to represent it using less space [1]. It is well-known that CPU speed has been growing faster than disk and network bandwidth during the last years. Therefore, reducing the size of the text, even at the expense of some CPU time, is useful because it reduces the I/O time needed to load it from disk or to transmit it thought a network.

Joining compression and block addressing indexes [12] improves the efficiency of that retrieval structures. Those indexes are smaller than standard inverted indexes because their entries do not point to exact word positions. Instead of that, entries point to those blocks where a word appears. This has a main drawback: some searches (i.e. phrase search) need traversing the text in the pointed blocks,

* This work is partially supported (for the Spanish group) by MCyT (PGE and FEDER) grant(TIC2003-06593), MEC (PGE and FEDER) grant (TIN2006-15071-C03-03), Xunta de Galicia Grant (PGIDIT05SIN10502PR) and (for the third author) by Millennium Nucleus Center for Web Research, grant (P04-067-F), Mideplan, Chile.

what usually implies decompressing the block. However, if the text is compressed with a technique that permits *direct search* in the compressed text, scanning the compressed blocks is much faster.

A good compressor for text databases has to join two main properties: *i)* to permit direct search into the compressed text by compressing the search pattern and then looking for this compressed version, and *ii)* to allow local decompression, which permits to decompress any portion of the compressed file without the need of decompressing it from the beginning. From the two main families of compressors (semistatic and adaptive compressors) only the semistatic ones join those two properties. Adaptive compressors, such as those from the well-known Ziv-Lempel family [15, 16] learn the data distribution of the source symbols as they compress the text and therefore, the mapping "source symbol $\leftrightarrow$ codeword" is adapted as compression progresses. Albeit there exist methods to search text compressed with adaptive compressors [14, 8], they are not very efficient. Decompression is usually needed during searches as the code given to a source symbol may vary along the text. Semistatic compressors (such as Huffman [10]) perform a first pass over the source text to obtain the distinct source symbols and to count their number of occurrences. Then, they associate each source symbol with a code that do not change during the second pass (where each source symbol is replaced by the corresponding codeword). Since the mapping source symbol $\leftrightarrow$ codeword does not vary, direct searching is allowed.

Classic Huffman is a well-known technique. It is a character-based method that generates an optimal *prefix* [1] coding. Unfortunately, it is not well-suited for text databases because of its poor compression ratio (around 60%). In [11], Moffat proposed using words instead of characters along with a Huffman coding scheme. As a result, compression ratio was reduced to around 25–30%. Moreover, using words instead of characters, gave the key to the integration of compression and text retrieval systems, since words are also the atoms of those systems.

Based on Moffat's idea, in [7] two word-based byte-oriented Huffman codes were presented. The first one, named Plain Huffman (PH) is a Huffman-based code that uses bytes instead of bits as target alphabet. By using bytes instead of bits, decompression speed was improved at the expense of compression ratio, which grew up to around 30–35%. The second compressor, named Tagged Huffman (TH), uses the first bit of each byte to mark the beginning of a codeword. Therefore, only 7 bits of each byte can be used to create the codewords, and a loss in compression of around 3 percentage points exists. However, since the beginning of a codeword can be recognized, random decompression is allowed and direct searches can be done by using a fast Boyer-Moore matching algorithm [2].

In [5, 4] we presented End-Tagged Dense Code (ETDC), a statistical semistatic technique that maintains the good capabilities of Tagged Huffman for searches while improving its compression ratio and using a simpler and faster coding scheme.

[1] In a prefix code, no codeword is a prefix of another, a property that ensures that the compressed text can be decoded as it is processed, since a lookahead is not needed.

In this paper, we present a modification over ETDC (see next section) that we call *Pair-Based End-Tagged Dense Code (PETDC)*, which improves its compression ratio (around 28–30%) by exploiting the co-occurrence of words in the source text. Moreover, PETDC permits direct searching the compressed text, as well as fast random decompression. The paper is structured as follows: In Section 2, ETDC is shown. Then PETDC is described in Section 3. In Section 4, empirical results measuring the efficiency of PETDC in compression ratio, compression and decompression speed are given. Finally, some conclusions end the paper.

2 Related Work: End-Tagged Dense Code

End-Tagged Dense Code (ETDC) [5, 4] is a semistatic compression technique, which is the basis of the new PETDC presented in this paper.

Plain Huffman Code [7] is a word-based Huffman code that assigns a sequence of bytes (rather than bits) to each word. In Tagged Huffman [7], the first bit of each byte is reserved to flag whether the byte is the first of its codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag bit is not useful by itself to make the code a prefix code. The tag bit permits direct searching the compressed text by just compressing the pattern and then running any classical string matching algorithm like Boyer-Moore [13, 9]. In Plain Huffman this does not work, as the pattern could occur in the text not aligned to any codeword [7].

Instead of using a flag bit to signal the *beginning* of a codeword, ETDC signals the *end* of the codeword. That is, the highest bit of any codeword byte is 0 except for the last byte, where it is set to 1.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits of each byte.

ETDC obtains a better compression ratio than Tagged Huffman while keeping all its good searching and decompression capabilities. On the other hand, ETDC is easier to build and faster in both compression and decompression.

In general, ETDC can be defined over symbols of b bits, although in this paper we focus on the byte-oriented version where $b = 8$.

Definition 1. *Given source symbols with decreasing probabilities* $\{p_i\}_{0 \le i < n}$ *the corresponding codeword using the End-Tagged Dense Code is formed by a sequence of symbols of b bits, all of them representing digits in base* 2^{b-1} *(that is, from 0 to* $2^{b-1} - 1$*), except the last one which has a value between* 2^{b-1} *and* $2^b - 1$*, and the assignment is done in a sequential fashion.*

That is, the first word is encoded as $\underline{1}0000000$, the second as $\underline{1}0000001$, until the 128^{th} as $\underline{1}1111111$. The 129^{th} word is coded as $\underline{0}0000000{:}\underline{1}0000000$, 130^{th} as $\underline{0}0000000{:}\underline{1}0000001$ and so on until the $(128^2 + 128)^{th}$ word $\underline{0}1111111{:}\underline{1}1111111$.

Note that the code depends on the rank of the words, not on their actual frequency. As a result, only the sorted vocabulary must be stored with the compressed text to allow the decompressor to recover the source text.

It is clear that the number of words encoded with 1, 2, 3 etc, bytes is fixed (specifically 128, 128^2, 128^3 and so on) and does not depend on the word frequency distribution. Generalizing, being k the number of bytes in each codeword ($k \geq 1$) words at positions i:

$$2^{b-1}\frac{2^{(b-1)(k-1)}-1}{2^{b-1}-1} \leq \ i \ < 2^{b-1}\frac{2^{(b-1)k}-1}{2^{b-1}-1}$$

will be encoded with k bytes. These clear limits mark the change points in the codeword lengths and will be relevant in the PETDC that we present in this paper.

But not only the sequential procedure is available to assign codewords to the words. There are simple *encode* and *decode* procedures that can be efficiently implemented, because the codeword corresponding to symbol in position i is obtained as the number x written in base 2^{b-1}, where $x = i - \frac{2^{(b-1)k}-2^{b-1}}{2^{b-1}-1}$ and $k = \left\lfloor \frac{\log_2\left(2^{b-1}+\ (2^{b-1}-1)i\right)}{b-1} \right\rfloor$, and adding 2^{b-1} to the last digit.

Function *encode* obtains the codeword $C_i = encode(i)$ for a word at the i-th position in the ranked vocabulary. Function *decode* gets the position $i = decode(C_i)$ in the vocabulary for a codeword C_i. Both functions take just $O(l)$ time, where $l = O(\log(i)/b)$ is the length in digits of codeword C_i. Those functions are efficiently implemented through just bit shifts and masking.

End-Tagged Dense Code is simpler, faster, and compresses 7% better than Tagged Huffman codes. In fact, ETDC only produces an overhead of about 2% over Plain Huffman. On the other hand, since the last bytes of codewords are distinguished, ETDC has all the search capabilities of Tagged Huffman code. Empirical comparisons between ETDC and Huffman codes can be found in [5, 4].

3 Pair-Based End-Tagged Dense Code

PETDC is a semistatic compressor based on ETDC. As in all semistatic compressors, a first pass over the source text is performed in order to gather the different words of the source text (vocabulary) and their number of occurrences. After that first pass, an encoding scheme is used to assign a codeword to each symbol in the vocabulary. A second pass over the source text replaces each source symbol by the codeword associated to it. In addition, the compression process ends by storing the vocabulary in such a way that, for each codeword, we can obtain its corresponding original symbol.

However, there are some differences between PETDC and other semistatic compressors such as ETDC, PH, etc. During the first pass, PETDC obtains an initial vocabulary by gathering the different words and their number of occurrences in the source text. Moreover, PETDC collects also all the different pairs

of words that appear adjacent in the source text and counts their number of occurrences. PETDC aims to take advantage of the co-occurrence of words in the text by including some pairs in the vocabulary (which is composed by both single-words and pairs). Its main idea is simple: In classic semistatic compressors, each symbol in the vocabulary has a unique codeword assigned by the encoding scheme (in this case, ETDC is used). Therefore, replacing two source words by a unique codeword during the second pass may need less bytes than replacing two single words by two codewords. Example 1 clarifies this situation.

Example 1. Let us compress the sequence of words: ADCBACDCCDABABACDC with ETDC, assuming that special bytes of only 3 bits are used, and that our words occupy 1 byte each. Therefore, the mapping word-codeword generated by the encoding schema is: $C \leftarrow \underline{1}00$, $A \leftarrow \underline{1}01$, $D \leftarrow \underline{1}10$, $B \leftarrow \underline{1}11$. Hence, all the words can be encoded with only one byte, and the size of the compressed text is 18 bytes. In this case, the vocabulary consists of only 4 words. As a result, the size of the compressed file is $18 + 4 = 22$ bytes.

Let us add to the vocabulary the most frequent pair of words 'BA', and compress the same text again. In this case, the vocabulary contains the symbols 'C', 'D', 'BA', 'A', and 'B', while the number of occurrences is 6, 4, 3, 2, and 0 respectively. Now the compressed text uses 15 bytes, and the vocabulary needs 6 bytes. Therefore, the compressed file occupies $15 + 6 = 21$ bytes. □

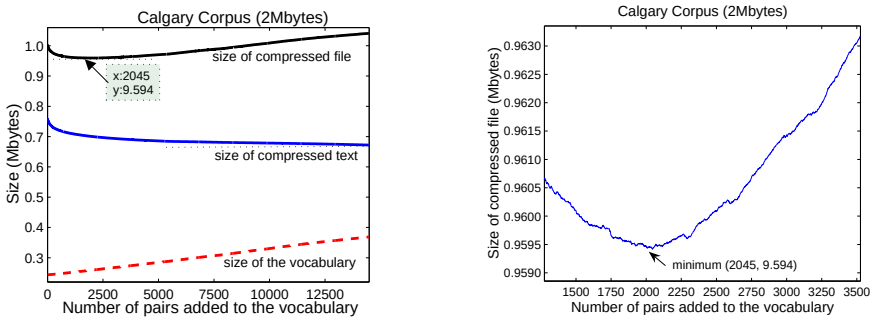
From Example 1 we show that processing some pairs of words as a unique source symbol reduces the size of the compressed text. However, as a drawback, the size of the vocabulary grows when a pair is added. Therefore, the existence of a trade-off between compressed text size and vocabulary size has to be taken into account when pairs are added to the vocabulary.

3.1 Deciding which pairs should be added to the vocabulary

Adding all the different pairs to the vocabulary is not a good idea because the vocabulary would grow too much. In Figure 1(a), we show the evolution of the size of a compressed file (as the sum of the size of the compressed data and the size of the vocabulary) depending on the number of pairs added. As expected, including the most frequent pairs in the vocabulary improves compression. However, at some point, the gain obtained by replacing two words by a unique codeword does not compensate the growth of the vocabulary size.

In Figure 1(b) it is shown that the previous curve has multiple local minima. This fact prevents us of looking for a heuristic to speed up the process by just breaking the addition of pairs to the vocabulary when the addition of a new pair worsens the compression. Instead of that, PETDC process all the pairs and applies a heuristic to determine which ones have to be added.

Used heuristic. Let us assume that a pair $\alpha\beta$, composed of two words α and β, is a candidate to be added to the vocabulary. Let us define f_x as the number of occurrences of a word or pair x. Let us also define C_x as the codeword that the



(a) whole process. (b) area close to maximum compression.

Fig. 1. Evolution of compressed file as pairs are added.

encoding scheme[2] assigns to x, and let $|C_x|$ be the length of that codeword. The heuristic is based on comparing the number of bytes needed to encode all the occurrences of α and β in the text in two cases: *i)* The pair is skipped ($skip_{bytes}$), and *ii)* the pair is added to the vocabulary (add_{bytes}). Once those values are computed, the pair $\alpha\beta$ is added to the vocabulary if $skip_{bytes} > add_{bytes}$ and skipped otherwise. Values $skip_{bytes}$ and add_{bytes} are given by the two following expressions:

$$skip_{bytes} = f_\alpha * |C_\alpha| + f_\beta * |C_\beta|$$
$$add_{bytes} = f_{\alpha\beta} * |C_{\alpha\beta}| + (f_\alpha - f_{\alpha\beta}) * |C'_\alpha| + (f_\beta - f_{\alpha\beta}) * |C'_\beta| + K$$

Where C'_α and C'_β are the codewords assigned to the words α and β assuming that the pair $\alpha\beta$ is added, and therefore their number of occurrences is $f_\alpha - f_{\alpha\beta}$ and $f_\beta - f_{\alpha\beta}$ respectively. The term 'K' is an estimation of the number of bytes needed to store any pair into the vocabulary. In general, $K = 5$.

Particular cases. There are two special situations that arise when pairs of words are considered:

- After adding a pair $\alpha\beta$ to the vocabulary it is necessary to ensure that any pair ending in α or beginning in β will not be included later. This happens because, by an efficiency issue, we do not store all the words that precede or follow any occurrence of $\alpha\beta$ in the text. As a result, given the text '$\gamma\alpha\beta\delta\alpha\mu$', adding the pair $\alpha\beta$ implies that the pairs $\gamma\alpha$, $\beta\delta$, and $\delta\alpha$ cannot be added to the vocabulary. This is done by just marking α as "disabled as first word of pair" and marking β as "disabled as last word of pair", and finally checking those flags before adding a pair to the vocabulary.
- Sequences of the same word: The appearance of sequences of the same word α such as $\alpha_1\alpha_2\alpha_3\alpha_4$ might lead us to count erroneously the number of occurrences of the pair $\alpha\alpha$. Note that $\alpha_2\alpha_3$ is not a valid pair if the pair $\alpha_1\alpha_2$ is chosen. To avoid this problem, when a sequence is detected we only count the occurrences of the pairs that start in odd positions.

[2] The codeword assigned to a word by ETDC depends only on the rank in the sorted vocabulary.

3.2 Data Structures and Compression Procedure

The data structures used by the compressor are sketched in Figure 2. There are two well-defined parts: *i)* Data structures that make up the vocabulary, and *ii)* data structures needed to hold the candidate pairs.

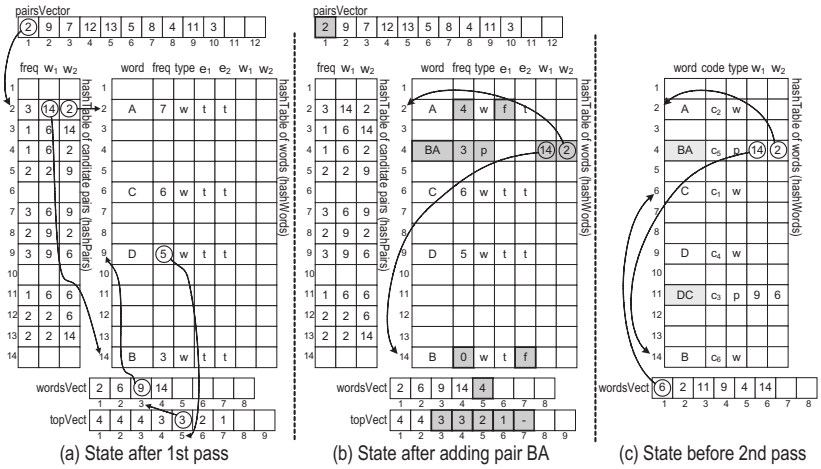


Fig. 2. Structures used in PETDC for text "ADCBACDCCDACADABABACDC".

- The vocabulary of the compressor consists of: A hash table used to locate a word quickly (*hashWords*) and two vectors: *wordsVect* and *topVect*. The hash table hashWords contains eight fields: *i) type* tells if an entry is either a word 'w' or a pair 'p', *ii)* if the entry has type 'w', *word* stores the word in ascii, *iii) freq* counts the number of occurrences of the entry, *iv-v)* e_1 *and* e_2 flag if the word is enabled to be the first or second component of a pair respectively, *vi-vii)* w_1 *and* w_2 store, for an entry of type 'p', pointers to the words that form the pair, and *viii) code* stores the codeword assigned to each entry of the vocabulary after the code generation phase.

  Vector *wordsVect* serve us to maintain the vocabulary sorted by frequency. Then, slot 1 of *wordsVect* points to the entry of *hashWords* where the most frequent word (or pair) in the source text is stored. Assuming that *wordsVect* is sorted decreasingly by frecuency, vector *topVect[f]* keeps track of the first entry in *wordsVect* whose frequency is f.
- Managing the candidate pairs includes also the use of two main data structures: *i)* A hash table *hashPairs* with fields *freq*, w_1, and w_2, used to give a fast access to each candidate pair, and *ii)* a vector *pairsVector* that maintains all the candidate pairs sorted, in the same way as *wordsVect*.

Compressing with PETDC. Compression consists of five main phases:

- *First pass along the text.* As shown, during this pass, PETDC obtains the different v single-words and the different p candidate pairs that appear in

the text. Moreover, it also counts their occurrences. The process costs $O(n)$, being n the number of words in the text. When the first pass ends, vectors *pairsVect* and *wordsVect* are sorted by decreasing frequency. Finally, *topVect* is initialized. Starting from element n downto 1, $topVec[i] = j$ if j is the first entry in $wordsVect$ such that $hashWords[wordsVect[j]].freq = i$. If $\nexists j$ such that $hashWords[wordsVect[j]].freq = i$, then $topVect[i] = topVect[i+1]$. The overall cost of this first phase is $O(n)+O(v\ log(v))+O(p\ \log(p))+O(v) = O(n) + O(p\ \log(p))$. Since $n \gg p$, we empirically proved that it costs $O(n)$.

- *Choosing and adding candidate pairs.* During this phase, $pairsVector$ is traversed ($O(p)$). A candidate pair $\alpha\beta$ is either added to the vocabulary or discarded, by applying the heuristic explained in Section 3.1. To compute that heuristic we need to know the current position[3] of α and β in the vocabulary to know the size of their respective codewords ($|C_x|$). We also need to assume that the pair $\alpha\beta$ is added, and we need to compute the new ranks of $\alpha\beta$, α, and β in the new ordered vocabulary. Since maintaining the vocabulary ordered upon inserting a pair is too expensive, we only maintain $topVect$ updated in such a way that, given a frequency f, $topVect[f]$ stores the rank, in a sorted vocabulary, of the first entry of frequency f. Then, being x an entry with frequency f_x, we estimate $|C_x|$ as $|C_{(topVect[f_x])}|$. It costs $O(F)$ time, where F is the frequency of the second most frequent entry of the vocabulary. Of course, $\alpha\beta$ is also inserted into $hashWords$ and into $wordsVector$. The overall cost of this phase is $O(p_a F + p) = O(p_a F)$, being p_a the number of pairs added to the vocabulary. Figure 2(b) shows the result of adding the pair "BA" to the vocabulary.
- *Code Generation Phase.* The only data structures needed in this phase are depicted in Figure 2(c). The vocabulary (with v' entries) is ordered by frequency and the encoding scheme of ETDC is used. Encoding takes $O(v')$ time. As a result, $hashWords$ will contain the mapping $entry_i \rightarrow code_i\ \forall i \in 1 \ldots v'$. The cost of this phase is $O(v' \log v')$.
- *Second pass.* The text is traversed again reading two words at a time and replacing source words by codewords. If the read pair $\alpha\beta$ belongs to $hashWords$ then the codeword $C_{\alpha\beta}$ is output and two new words $\gamma\delta$ are read. Otherwise C_α is output and the only the following word γ is read to form a new pair $\beta\gamma$. This phase takes $O(n)$ time.
- *storing the vocabulary.* As in ETDC, the vocabulary is stored along with the compressed data to permit decompression. A bitmask is used to save the type of entry. Then the v' entries of the vocabulary follow that bitmask. A single-word is written in ascii (ending in '\0'). To store a pair $\alpha\beta$ we write the relative positions of α and β in the vocabulary (encoded with the on-the-fly $C_w = getCode(i)$ function used in [3] in order to save space). Finally, the whole vocabulary is encoded with character-based Huffman.

Decompressing text compressed with PETDC. Decompression starts by loading the vocabulary into a vector. Then each codeword is easily parsed due

[3] Using the encoding scheme of ETDC, we can compute in $O(\log i)$ time the codeword $C_i = getCode(i)$ where i is the rank of a word w_i in the vocabulary.

to the flag bit that marks the end of a codeword. For each codeword C_i, the function $i = decode(C_i)$ [3] is used to obtain the entry i that contains either the word or the pair associated to C_i.

Searching text compressed with PETDC. In text compressed with PETDC, a word α can appear alone or as a part of one or more pairs $\alpha\beta$, $\gamma\alpha$,... Therefore searches will usually imply using a multi-pattern matching algorithm. When we load the vocabulary, we can easily generate for each single-word α, a list with the codewords of the pairs in which it appears. After that, an algorithm from the Boyer-Moore family such as *Set Horspool* [9, 13] is used to search for those codewords and for the codeword C_α.

4 Empirical Results

We compare PETDC against other semi-static word-based compressors such as PH and ETDC and against two well-known compressors such as Gnu *gzip*[4], a Ziv-Lempel compressor and Seward's *bzip2* [5], a compressor based on the Burrows-Wheeler transform [6]. We used some large text collections from TREC-2 [6], namely AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as some from TREC-4: Congressional Record 1993 and Financial Times 1991 to 1994. We used the spaceless word model [7] to create the vocabulary; that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

Our first experiment is focused in comparing compression ratio. Table 1 shows in the two first columns the corpora used and their size. Columns three to six gives information about applying PETDC such as the number of candidate pairs, the number of occurrences of the most frequent pair, the number of pairs added, and the number of entries (pairs and words) of the vocabulary. The last five columns show compression ratio (in percentage) for the compressors used. It can be seen that PETDC improves the compression of PH and ETDC by around 3 and 4 percentage points respectively. Gaps against *gzip* grow up to $6-7$ percentage points. Finally, PETDC is overcome by *Bzip2* in around $1-3$ percentage points.

We focus now on comparing PETDC against other alternatives in compression and decompression speed. An isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 512 MB single-channel DDR-400Mhz was used in our tests. It ran Mandrake Linux 9.2. The compiler used was gcc version 3.3.1 and `-O9` compiler optimizations were set. Time results measure CPU user time in seconds. As it is shown in Table 2, PETDC pays the extra-cost of managing pairs during compression, being around 2.5 times slower than ETDC and PH, and around $1.5-2$ times slower than *gzip*. However, it is much faster than *bzip2*. In decompression, the extra-cost of PETDC consists only in processing

[4] `http://www.gnu.org`.
[5] `http://www.bzip.org`.
[6] `http://trec.nist.gov`.

Corpus	Size (bytes)	cand. pairs	highest freq.	pairs added	entries vocab	Compression ratio (%)				
						PETDC	PH	ETDC	gzip	bzip2
Calgary	2,131,045	53,595	2,618	4,705	35,700	41.31	42.39	43.54	37.10	29.14
FT91	14,749,355	239,428	18,030	12,814	88,495	30.72	33.13	34.02	36.42	27.06
CR	51,085,545	589,692	70,488	38,090	155,803	27.66	30.41	31.30	33.29	24.14
FT92	175,449,235	1,592,278	222,505	113,610	397,671	28.46	31.52	32.33	36.48	27.10
ZIFF	185,200,215	2,585,115	344,262	194,731	418,132	29.04	32.52	33.41	33.06	25.11
FT93	197,586,294	1,629,925	236,326	124,547	415,976	27.79	31.55	32.43	34.21	25.32
FT94	203,783,923	1,666,650	242,816	123,583	418,601	27.78	31.50	32.39	34.21	25.27
AP	250,714,271	1,574,819	663,586	118,053	355,674	28.80	31.92	32.73	37.32	27.22
ALL_FT	591,568,807	3,539,265	709,233	314,568	891,308	27.80	31.34	32.22	34.94	25.91

Table 1. Compressing with PETDC and comparison in compression ratio with others.

the bitmask in the header of the vocabulary file and rebuilding the pairs from the pointers to single-words. Therefore, the loss of speed against PH, ETDC, and *gzip* is small (under 20%), and PETDC becomes around $6-8$ times faster than *bzip2*.

Corpus	Compression time (seconds)					Decompression time (seconds)				
	PETDC	PH	ETDC	gzip	bzip2	PETDC	PH	ETDC	gzip	bzip2
Calgary	0.59	0.33	0.37	0.26	0.88	0.06	0.06	0.06	0.04	0.32
FT91	3.07	1.48	1.47	1.71	6.33	0.30	0.25	0.25	0.20	2.24
CR	8.85	3.97	4.01	5.83	21.26	0.78	0.66	0.65	0.64	7.52
FT92	33.24	13.86	13.85	20.28	76.47	2.90	2.39	2.50	2.36	29.42
ZIFF	33.47	13.84	13.87	20.20	79.70	2.83	2.35	2.32	2.20	27.00
FT93	35.81	15.61	15.30	21.34	80.21	3.17	2.73	2.75	2.62	32.58
FT94	37.11	15.84	15.73	22.08	88.96	3.33	2.81	2.87	2.63	33.78
AP	48.50	20.06	20.26	30.66	105.48	4.11	3.41	3.41	3.43	39.42
ALL_FT	113.26	45.29	45.12	64.82	254.45	9.63	8.00	8.12	7.49	89.67

Table 2. Comparison in compression and decompression time.

5 Conclusions

We have presented a new semistatic pair-based byte-oriented compressor that we named *Pair-Based End-Tagged Dense code(PETDC)*. It takes advantage of using both words and pairs of words (exploiting the co-occurrence of words) to improve the compression obtained by similar word-based semistatic techniques such as PH or ETDC.

Dealing with pairs has a cost in compression speed (PETDC is around 2.5 times slower than ETDC) and in decompression (PETDC is 20% slower than ETDC). However, the new technique is able to reduce English texts to 27–28% of its original size (over 3 percentage points better than PH). Moreover, it maintains the ability of performing *direct searches* and *random decompression* of a portion of the text.

To sum up, PETDC is a technique well-suited to use in Text Databases due to its good compression ratio and decompression speed, as well as for its good search capabilities. The main drawback with respect to others might be its medium compression speed. However, in a text retrieval scenario a medium compression speed is only a minor problem since compression is done only once.

Acknowledgments. We want to thank Àngel Yàñez Miragaya for his help in the implementation of PETDC.

References

1. T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. P.Hall, 1990.
2. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.
3. N. Brisaboa, A. Fariña, G. Navarro, and José R. Paramá. Simple, fast, and efficient natural language adaptive compression. In *Proceedings of the 11th SPIRE*, LNCS 3246, pages 230–241, 2004.
4. N. Brisaboa, A. Fari na, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 2006. To appear.
5. N.R. Brisaboa, E.L. Iglesias, G. Navarro, and José R. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th ECIR*, LNCS 2633, pages 468–481, 2003.
6. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
7. E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.
8. M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. In *Proceedings of the 27th ACM-STOC*, pages 703–712, 1995.
9. R. N. Horspool. Practical fast searching in strings. *SPE*, 10(6):501–506, 1980.
10. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Eng.*, 40(9):1098–1101, 1952.
11. A. Moffat. Word-based text compression. *SPE*, 19(2):185–198, 1989.
12. G. Navarro, E.S. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *IR*, 3(1):49–77, 2000.
13. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical online search algorithms for texts and biological sequences*. Cambridge Univ. Press, 2002.
14. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proceedings of the 11th CPM*, LNCS 1848, pages 166–180, 2000.
15. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TIT*, 23(3):337–343, 1977.
16. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 24(5):530–536, 1978.