Saving Energy in Text Search Using Compression

Javier Mancebo, Coral Calero, Félix García
Institute of Technology and Information Systems
University of Castilla-La Mancha
Ciudad Real, Spain
e-mail: {Javier.Mancebo, Coral.Calero, Felix.Garcia}@uclm.es

Nieves R. Brisaboa, Antonio Fariña, Óscar Pedreira
CITIC Research Center, Faculty of Computer Science,
University of A Coruña
A Coruña, Spain
e-mail: {brisaboa, antonio.farina, oscar.pedreira}@udc.es

Abstract— **The widespread use of text databases and their exponential growth has boosted the interest in developing text compression techniques. These techniques aim at representing text collections using less space, but also at efficiently processing them by providing functionalities such as searching for words and phrases in the compressed text or decompressing just a portion of the text. In this paper, we study compression techniques from the perspective of the savings obtained in energy consumption when searching directly within the compressed text. Our results show that the use of text compression techniques can have an important influence on energy efficiency when the text is processed, for example, for word or phrase searches. Our evaluation compares the energy consumption of both compressed and uncompressed text searches, and the results show that compressing text databases can lead to important energy savings (around 50%).**

Keywords – Text compression; End-Tagged Dense Codes; Energy efficiency; Green software.

I. INTRODUCTION

In recent years, the amount and size of text collections has increased considerably. Although the capacity of new devices to store such a large amount of data is growing quickly, the rate of generation and growth of text collections is even higher. Compression techniques raised as a natural solution to reduce space needs and to save transmission times [1]. However, in a text database, not only space matters but also the ability to efficiently perform queries and to retrieve any part (e.g., decompressing a relevant document) of the whole collection [2][3]. This led to the creation of word-based text compression techniques that not only allowed to reduce the size of the text database to around 30%-35% of the original size but also to efficiently search for any word or phrase directly in the compressed text, avoiding the need for decompressing before searching.

Typically, if one aims at providing efficient access to a document collection, the usual choice is to provide indexed searches that require an additional indexing structure. One could opt for a full-text inverted index that tracks the exact positions where each word occurs within the collection. This index would occupy around 30-40% of the size of the original collection [4], but efficiently supports both word and phrase searches. To save space, document- or even block-addressing inverted indexes can be used instead [4]. For each word, these indexes keep a list of documents/blocks where the word occurs, hence allowing us to directly filter out the documents/blocks where a word occurs. However, solving phrase searches still requires to sequentially scan those documents/blocks containing the words than compose such phrase to verify that they occur in adjacent positions. Consequently, the raise of text compressors that permitted to perform searches directly within the compressed text even faster than when those searches were performed over plain text, boosted the performance of block-addressing inverted indexes [4]-[6] and allowed us to represent the compressed text plus the index in just around 35-45% of the size of the original collection [3] and typically still permit to solve queries within the range of 1-100 milliseconds. In practice, the larger the block size chosen for the block-addressing inverted index, the smaller the index and the longer blocks have to be sequentially traversed at query time.

In this paper, we do not tackle indexing, but we consider a new aspect of text compression techniques. We focus on the energy consumed during the compression process and, more importantly, at search time, where we compare the energy used to perform online searches on both compressed and uncompressed text.

As a matter of fact, the energy consumption of Information Technology (IT) solutions has become in recent years an important concern, and, in particular, software development and maintenance with a "green" perspective in mind [7][8]. If we focus on text databases, energy consumption is a particularly important aspect due to the large number of searches that can be performed on it. Therefore, even small savings in the amount of energy required to perform a single search can be transformed into huge savings when all the searches performed are considered.

We provide experimental results in which we study the energy required when a text collection is compressed and, more importantly, we measure the energy consumption involved when searching both compressed and uncompressed text. Without losing generality, we focus our experimental evaluation on a given compressor, namely End-Tagged Dense Code (ETDC) [6], since it is the best compressor for representing text databases in a compressed form [3][4] while still being able to search for words or sentences in the compressed text without previously decompressing it.

 ISBN: 978-1-61208-751-1

All our energy consumption measurements were carried out by using an Energy Efficiency Tester (EET) device [9].

The remainder of this paper is organized as follows: Section II briefly presents basic concepts on text compression and describes the ETDC technique. Then, it briefly discusses the search mechanism used to efficiently perform searches for words in both compressed and uncompressed text. In Section III, we explain the main aspects of software sustainability, as well as the framework used to measure software energy consumption. Section IV presents the experimental design framework, and provides the results obtained from our preliminary study. Our main conclusions and future work are discussed in Section V.

II. BACKGROUND

A. Basic concepts on text compression

Word-based compressors represent each word in the text with a code (also called *codeword*) and achieve compression by assigning shorter codes to the most frequent words. By typically using either Huffman [10] (or Dense [6]) coding, the codewords assigned become prefix-free codes. This means that no codeword can be a prefix of a longer codeword and ensures that decoding any codeword can be efficiently performed without the need of any look-ahead.

Huffword [2] is a Huffman-based compressor where each word is given a *bit-oriented* code, that is, the code is a sequence of bits. It yields strong compression ratios, around 25% in English texts, but being bit-oriented, decompression and searches within the compressed text are slow. Plain Huffman [5] assigns *byte-oriented codes* to each word, that is, each code is a sequence of bytes. By using codes made of bytes instead of bits, compression ratios worsen to around 30%, yet byte-wise decoding becomes much faster than that of Huffword, and searches (using a variant of the Shift-Or string matching algorithm [5]) are also largely sped up.

However, Tagged Huffman Codes [5] brought the most important break-through regarding search efficiency within the compressed text. The difference with Plain Huffman is that the first bit of each byte is used to mark if that byte is the first byte of a codeword. In this way, Tagged Huffman Codes became suffix-free (a codeword cannot be a suffix of a longer codeword), what allowed to use the fastest Boyer-Moore [11] string matching algorithms to directly search in the compressed text [5].

By reserving 1 bit of each byte, the compression of Tagged Huffman worsened slightly (compression ratios around 35%) with respect to Plain Huffman. Yet, searches were largely boosted. Indeed, searches for either words or phrases within text compressed with Tagged Huffman can be up to eight times faster than the same searches in uncompressed text [5]. Furthermore, since the beginning of any codeword is now distinguishable, Tagged Huffman also gained self-synchronization capabilities, that is, we can access any byte of the compressed text and start decompression from there on without the need for synchronization from the beginning of the text. Even considering the loss of compression ratio with respect to Plain Huffman, its improved search capabilities promoted Tagged Huffman as the best choice to compress text databases until the proposal of Dense Codes [6].

The ETDC uses the first bit of each byte to mark the last byte of each codeword instead of the first one, as in Tagged-Huffman. This simple idea makes ETDC a prefix-free coding (without the need of applying Huffman coding), and allows ETDC to have all the same interesting properties of Tagged-Huffman but achieving compression ratios closer to those of Plain Huffman (around 31%) which pushed ETDC as the best compressor for text datasets [4]. In summary, the main strengths of ETDC are its good compression, fast compression and decompression procedures, as well as random decompression capabilities (self-synchronization), and the ability to directly search within the compressed text using Horspool algorithm [12].

For these reasons, our empirical evaluation studies, without loss of generality, the energy savings obtained when searching within text compressed with ETDC.

B. End-Tagged Dense Code: compression and search

ETDC [6] is a well-known two-pass word-based byte-oriented statistical compressor. As a two-pass compressor, it processes the source text twice at compression time. A first pass over the text is performed in order to gather both the different words/symbols (the set of different symbols is typically known as the vocabulary of symbols) and their frequency to make up a model of the original text. After that, the vocabulary of symbols is sorted by frequency, and then, as a statistical compressor, ETDC performs a coding stage where shorter codewords are given to the most frequent symbols. As a result of the coding stage, each symbol is associated a unique codeword. ETDC is a byte-oriented compressor, which means that codewords are variable-length sequences composed of 1, 2, or more bytes. The exact encoding mechanism of ETDC will be explained below. After the coding stage, a second pass over the original text is performed again, and, for each symbol of the original text ETDC outputs its corresponding codeword to a new file (compressed-file), hence creating the compressed representation of the original text. In addition, the correspondence symbol⟷codeword (i.e., the list of symbols sorted by frequency in the case of ETDC) must also be kept along with the compressed file (header-file) to allow a further decompression. The overall compression procedure is depicted in Fig. 1. Note that, by replacing the most frequent symbols by the shortest codewords, compression is obtained. In addition, since each source symbol is always replaced by the same codeword, when performing searches for a given word/symbol we can just obtain the codeword associated to such word and then look for the actual positions where the corresponding codeword occur within the compressed file. By using Horspool algorithm [12], searches are efficiently performed [6].

C. Encoding and decoding procedures in ETDC

The encoding procedure of ETDC is very simple [6]. In this case, given a symbol ranked at position i in the

vocabulary of words (decreasingly sorted by frequency) ETDC will assign to it a codeword c_i composed of 1 or more bytes (recall ETDC generates byte-oriented codewords).

One of the key-features of ETDC is that it marks the last byte of each codeword with a special flag. ETDC reserves the first bit of each byte from a codeword to mark if that byte is the last byte or not, i.e., the first bit of the last byte of a codeword is set to 1, and the first bit of the remaining bytes is set to 0. Therefore, one-byte codewords will have the form 1xxxxxxx; two-byte codewords have the form 0xxxxxxx:1xxxxxxx; three-byte codewords follow the pattern 0xxxxxxx:0xxxxxxx:1xxxxxxx; and so on. Note that basically, the numerical byte-values of the ending byte of any codeword are within the range [128,255], whereas the other bytes have values within the range [0,127].

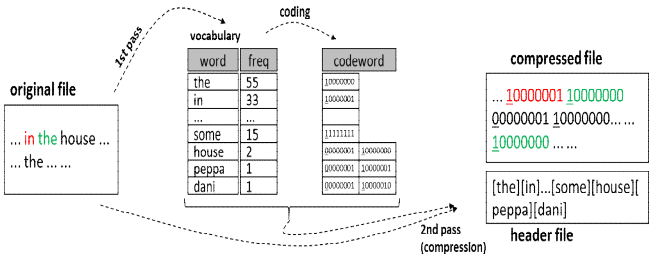

Figure 1. Compression process in ETDC

By marking the last byte of each codeword, ETDC becomes a prefix-free coding. Therefore, despite other well-known Huffman-based compressors [5][10] that reserve some bit-combinations to ensure that the final codewords own the prefix-free property, ETDC can use all the possible bit-combinations of each codeword byte. Actually, the codeword assignment in ETDC is done in a completely sequential fashion (considering the remaining 7 bits of each byte) and does not depend on the actual frequency value of the words, but only on their rank within the sorted vocabulary. The codeword assignment to words decreasingly sorted by frequency is done as follows:

- 1-byte codewords are given to the first 128 words in the vocabulary, i.e., the most frequent word receives codeword 10000000, the second word in the vocabulary is given codeword 10000001, and so on until the word ranked at position 128, which is assigned codeword 11111111. Note that the initial bit is 1 to flag that it is an ending codeword byte, and the remaining seven bits are assigned sequentially from 0000000 to 1111111.
- A 2-byte codeword is given to the next 128x128 words; i.e., word ranked at position 129 receives codeword 00000000:1000000; next word is given codeword 00000000:1000001; and so on until the word ranked at position $128+128^2$ which is given codeword 01111111:11111111.
- A 3-byte codeword is given to the next 128^3 words, from codeword 00000000:00000000:10000000 to codeword 01111111:01111111:11111111.
- If required, 4-byte codewords would be assigned to the next 128^4 words in the same way, and so on.

Codewords are sequentially assigned to words during compression. However, we can know at any moment which codeword corresponds to a given word. This operation is called *encode*, and allows us to search for any word or phrase in the compressed text. We can also know at any moment the word corresponding to a given codeword. This operation is called *decode*, and allows decompressing the text starting at any position.

D. *Searching: the Horspool algorithm*

Pattern-matching algorithms aim at efficiently finding the positions of a text in which a given *pattern* (a sequence of symbols, for example, a word) appears. A sequential brute-force search would be a simple but very inefficient solution.

The Horspool algorithm [12] searches for the pattern in a sequential fashion but skips parts of the text during the search. The algorithm uses a *search window* of the same size as the pattern we are searching for (see Fig. 2). This search window is moved along the text during the search. At each step, the text under the window is compared with the pattern. If they are equals, the algorithm reports a new occurrence of the pattern. If they are not, the algorithm moves the window to a new position, trying to skip as much text as possible. I.e., if β is the last symbol in the pattern, the skip distance is the number of symbols from the last appearance of β in the pattern to the end of the pattern. If the last symbol only appears in that last position, the skip distance is the size of the pattern. The efficiency of Horspool is given by its capacity to skip large portions of the text. Therefore, the larger the alphabet of symbols, and the larger the pattern, the more chances to skip text, so the more efficient the algorithm will be.

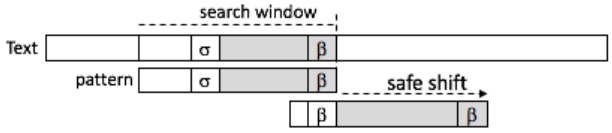

Figure 1. Horspool search algorithm scheme.

Although the Horspool algorithm was thought to be applied in plain text, it can also be applied to text compressed with ETDC. The result of applying this algorithm in text compressed with ETDC is that the searches in the compressed text are much more efficient than searches in the original plain text [5][7].

III. SOFTWARE SUSTAINABILITY

Recent research focused on the proper use of resources required by software has emerged in the area of software sustainability [13]. This resulted in more sustainable and environment-friendly software. According to Calero and Piattini [13], there are three dimensions of Software Sustainability: (i) Human sustainability, which analyzes how software development and maintenance can affect the sociological and psychological aspects of the people

 ISBN: 978-1-61208-751-1

involved; (ii) Economic sustainability, which is related to how software lifecycle processes protect stakeholders' investments, ensuring benefits and reducing risks; and (iii) Environmental sustainability, which studies how the development and maintenance, and the use of software affects the use of resources and the energy consumption. This dimension is also known as Green Software.

Our work focuses on the Environmental sustainability dimension. In particular, we analyze how the usage of the ETDC as a text compression technique reduces the amount of energy needed to perform direct searches, compared to the energy needed when directly searching within plain text. To do that, we use FEETINGS (Framework for Energy Efficiency Testing to Improve eNvironmental Goals of the Software) [9], whose objective is to measure and analyze the energy consumed by a software product when it is executed in a computer. This framework is divided into two main components:

- An Energy Efficiency Tester (EET), the hardware device that measures the energy consumption of a software product during its execution. The EET is composed of different sensors that support the measurement of three different hardware elements: processor, hard disk, and graphics card. Furthermore, two additional external sensors quantify the total power consumption of the computer, and that of the monitor connected to the computer (Device Under Test or DUT) where the software is executed.
- A Software Energy Assessment (SEA) application, which automatizes the processing, the analysis, and the visualization of the data collected by the EET.

IV. ETDC Energy Consumption Evaluation

A. Experimental design framework

We have carried out a set of experiments to evaluate whether the use of ETDC as the compression method to deal with a text collection not only leads to improvements in space and search times but also entails savings on the amount of energy consumed to perform searches. In addition, we also include results showing the time and energy consumption required to compress such text dataset.

We have used a text collection from [6] composed of a small text named Calgary [14], and several text collections from TREC-2 and TREC-4 [15] (AP Newswire 1988, Ziff Data 1989-1990, Congressional Record 1993, and Financial Times from 1991 to 1994). The size of the dataset is around 1,030 MiB, and when processed with ETDC, the vocabulary obtained has 886,190 different words.

Aiming at performing online searches for single-word patterns, we considered the vocabulary from the collection and we randomly chose words (we assume words are sought with uniform probability by following the same model in [5]) to make up three sets of patterns with varying length. These sets of patterns contain respectively 10 words whose length is 5, 10, and >10 characters. Note that, for this study, we did not filter those words out by frequency.

All our experiments were run on a DUT connected to the EET that was connected to an LCD monitor Philips 170S6FS. The DUT specifications are: (a) Asus M2N-SLI Deluxe motherboard; (b) AMD Athlon tm 64 X2 Dual Core 5600+ 2.81 GHz processor; (c) 4 modules of 1GB DDR2 533MHz RAM memory; (d) Seagate barracuda 7200 500Gb hard disk; (e) GPU Nvidia XfX 8600GTS; and (f) AopenZ350-08Fc 350 W Power supply. It runs Linux Mint 18.3 Cinnamon 32 bits, and the compiler used was gcc version 5.4.0.

We present two main experiments. First, we focus on searches and compare both the performance and the energy consumption obtained when we search for the ten words in each of our query sets, using Horspool algorithm, both over the ETDC-compressed representation of the dataset, and over its plain/original version. Finally, we complete our study showing the amount of energy consumed to compress the dataset with ETDC.

Each experiment was repeated 20 times and then averaged. Being a controlled test environment, 20 measurements are usually a sufficient sample size to mitigate the impact of outliers (such as energy consumption devoted to operating system tasks). Therefore, our time results are presented as average running times, and our energy consumption data are shown in average watt second (W·s). In practice, we will show four different energy-consumption values corresponding to the Hard Disk Drive (HDD), the graphics card (GPU), the processor (CPU), and the Power Supply Unit (PSU), which indicates the total energy consumption of our system.

We performed two-sample t-tests on the measurements we obtained from the experiments, assuming the variance of the samples obtained from measurements in plain text and ETDC are not equal. The test revealed that the means obtained in the two samples are different with a confidence of 99% ($\alpha = 0.005$).

B. Plain text searches vs. searches over text compressed with ETDC

In Table I, we show both the average time and the average energy consumption corresponding to searches performed over the original text and over the text compressed with ETDC. Recall that we use the three query sets discussed above corresponding to ten patterns of length 5, 10, and >10 characters, and our times include the overall time corresponding to ten Horspool searches for those ten patterns.

TABLE I. Time and energy consumption results when performing 10 searches with Horspool algorithm over both text compressed with ETDC and over uncompressed text

	Uncompressed Text					Text compressed with ETDC				
Pattern length	time (s)	Energy consumption (W·s)				time (s)	Energy consumption (W·s)			
		HDD	GPU	CPU	PSU		HDD	GPU	CPU	PSU
5	28.78	470.79	43.27	153.39	4120.08	14.04	230.18	20.53	62.71	1817.83
10	25.17	411.43	36.86	132.29	3647.17	13.00	212.79	19.48	56.50	1684.84
>10	23.87	390.89	34.79	126.29	3465.62	13.54	221.70	18.66	56.35	1771.77

TABLE II. TIME AND POWER MEASURED BY THE EET. POWER VALUES ARE OBTAINED AS THE ENERGY CONSUMPTION VALUES FROM TABLE 1 DIVIDED BY THE RUNNING TIME (IN SECONDS).

	Uncompressed Text					Text compressed with ETDC				
Pattern length	time (s)	Power (watt)				time (s)	Power (watt)			
		HDD	GPU	CPU	PSU		HDD	GPU	CPU	PSU
5	28.78	16.36	1.50	5.33	143.16	14.04	16.39	1.46	4.47	129.48
10	25.17	16.35	1.46	5.26	144.90	13.00	16.37	1.50	4.35	129.60
>10	23.87	16.38	1.46	5.29	145.19	13.54	16.37	1.38	4.16	130.85

Note that, in Table II, the HDD and GPU consumptions do not vary in the uncompressed and compressed representations. This permits us to conclude that, as expected, it is more energy efficient to perform searches over text compressed with ETDC than over uncompressed text. In particular, we can also see that the most important energy savings (in percentage) are obtained in the processor, followed by the total consumption values drawn by the PSU, and finally the HDD and the GPU are the elements where less savings are reached.

Considering search performance, as expected from [6], the search time is also lower in the compressed scenario (in practice, our compressed searches require from 43% to 51% less time). When we compare how the search times vary with respect to the pattern length, we can see that the longer the pattern, the faster the search is performed, and consequently less energy is required. This is expected since longer patterns lead to longer jumps during the left-to-right traversal in Horspool algorithm.

Even though it could seem rather unexpected, the search times over text compressed with ETDC only marginally depend on the pattern length. Yet, this is true. Given a pattern P, note that on compressed searches we do not search for pattern P, but for the codeword associated to P. Yet, in our dataset, all the words are given codewords of 1, 2, or 3 bytes (recall more frequent words are given shorter codewords). Consequently, the longest "*shift*" during the left-to-right traversal in Horspool will be of only 3 bytes. The average codeword length is 2.9, 3.0, and 3.0 respectively for the patterns in our query sets with patterns of length 5, 10, and >10 characters. This would explain that searches for patterns of length 5 are slightly slower. Yet, we would expect that search performance on patterns of length 10 and >10 should be similar.

In Table II, we have divided the total energy consumption values shown in Table I by the time required to run each experiment. This gives us power values for each component that are independent on the time needed to complete each run. It is interesting to see that both the HDD and the GPU (as expected) have rather constant power needs in both the compressed and the uncompressed scenarios. They require around 16.3 and 1.5 watt, respectively. However, the processor power decreases considerably in the compressed scenario. Not only the searches over compressed text perform faster, but the CPU requires also around 20% less power. On the one hand, Horspool benefits from a lower probability of match between the last character of the pattern, and the rightmost character on the sliding window of the text, and such probability is lower on text compressed with ETDC than on uncompressed text. In [16], it was shown that those probabilities of match are, respectively, around 1/119=0.008 in ETDC and 1/19.3=0.052 in plain text. This is due to the fact that we can find any of the 256 possible combinations of a byte both the compressed text and in the search pattern for ETDC, whereas in plain text less than 100 different byte values are used ([A-Z], [a-z], [0-9], and punctuation symbols). On the other hand, due to that lower probability of match in ETDC, Horspool algorithm wastes much less time comparing (right-to-left) the text and the pattern, and, consequently, most of time is devoted to the main (left-to-right) shift-loop. We conjecture that this makes the execution pipeline simpler and more predictable and reduces the power required by the processor.

C. Compressing text with ETDC

As shown above, searching within text compressed with ETDC is more energy efficient than performing searches over uncompressed text. In this section, we also take into account the energy consumption involved when compressing the original text. Table III shows the average time (for 20 repetitions), and the average energy consumption (and power) required to compress our dataset with ETDC.

TABLE III. COMPRESSION WITH ETDC: TIME AND ENERGY ONSUMPTION.

time (s)	Energy consumption (W·s)				Power (watt)			
	HDD	GPU	CPU	PSU	HDD	GPU	CPU	PSU
53.24	876.88	78.94	320.71	6867.21	16.47	1.48	6.02	128.99

As in Table II, we already observe that the power required by both the HDD and the GPU remain rather constant (around 16.5 and 1.5 watt, respectively). The reason is that those consumptions are close to the basal consumption of those devices in the computer. However, the power required by the CPU grows clearly with respect to the values obtained at search time. The CPU uses around 6 watts, whereas for searches over uncompressed text only around 5.3 watt were needed, and those values decreased to around 4.3 watt for searches over text compressed with ETDC. Since the power needs grow, and from the fact that compression takes more time than performing searches, the overall energy consumption increases accordingly.

If we informally analyze the obtained data, we can see that the energy consumption devoted to compress the original text is equivalent to the consumption of performing twenty searches within the uncompressed text. Similarly, from the point of view of energy efficiency, if one is going to search for more than 40 words over a whole large text dataset, it would compensate to keep the text compressed with ETDC, and to perform the searches over such compressed text. This typically can lead to savings around 50% with respect to performing the same searches over the original plain text.

D. Threats to validity

The study presented in this work has some limitations that should be taken into account to understand to what extent the results are valid. According to the classification of threats discussed by Wohlin et al. [17] we can identify the following threats:

 ISBN: 978-1-61208-751-1

- *Threats to Construct Validity*: the main threat relates to whether the obtained energy consumption measurements are correct. In our case, we have overcome this threat by using the EET device [9]. This device has been validated and compared with another measuring device in [18]. The energy consumption results obtained by both devices were similar. In addition, EET has previously been used in other similar measurements. Additionally, we have used different patterns of varying length. This fact permitted us to analyze how the length of the pattern influences energy consumption.
- *Threats to Internal Validity*: these threats are mainly related to the configuration of the DUT in which the measurements are made. As it is evident, if we had used a different DUT, we would have obtained different data. However, we believe that, even though the absolute values of the measurements could have varied, the energy consumption relations/conclusions would still remain. Note also that the state of the Operating System (e.g., existence or not of other running tasks) could also be a relevant factor that should be considered. To overcome this possible issue and obtain more stable measurements, each measurement was repeated 20 times.
- *Threats to External Validity*: We have used our own EET as the tool for measuring energy consumption. As indicated above, this device is able to obtain exact measurements of the energy consumed by different hardware elements. Obviously, the measurements obtained are specific for our EET and may differ if we use other mechanisms, such as energy estimation or other devices. Nevertheless, our EET was designed for the actual measurement of different hardware components when a given software is running, and we consider the results obtained to be correct.
- *Threats to Conclusion Validity*: We have analyzed energy consumption when searching a compressed text only by the ETDC algorithm. Therefore, the results obtained cannot be assumed for other text compression algorithms. Yet, as discussed in the future work section, we would expect a rather similar behavior.

V. Conclusions and Future Work

Compression methods have become a widely-used resource nowadays. This is due to the large amount of data that is generated and that must be stored. Undoubtedly, compression permits to store those data within less space. In the scope of text databases, where the ability to perform searches and to retrieve some parts of the text collection is of major interest, End-Tagged Dense Code [6] becomes one of the best compression alternatives due to its reasonable compression ratios (around 30-35%), fast compression and decompression processes, the possibility of performing direct searches on the compressed text very efficiently using Horspool algorithm, and by allowing random decompression (i.e., starting decompression from any random offset of the compressed file).

In this paper, we have also considered a fundamental concern such as the amount of resources used (by means of energy consumption) when performing searches. We have compressed a large text database with ETDC and then performed queries both over the original uncompressed text and over the compressed counterpart. Our results show that compression not only reduces space and searching time, but also leads to less energy being consumed. On the one hand, one could expect that since a faster algorithm requires power during a shorter amount of time, the overall energy consumption would decrease. Our results clearly confirm that. On the other hand, we found a rather unexpected result: the same Horspool algorithm, running over compressed data also required less CPU power than when it ran over uncompressed data.

As future work, we want to extend our study to include other well-known compression techniques for text databases that own similar features to those in ETDC. Among them some good candidates are Tagged and Plain Huffman, or the Restricted Prefix Byte Codes [19]. In this way, we will be able to know which of these techniques requires less energy in compression time, and which one provides more efficient searches in terms of energy. In addition, we intend to expand the study by analyzing CPU utilization when performing text searches (with and without compression), and to determine the relationship between CPU utilization and energy consumption.

Once it is clear that compression permits to reduce energy utilization at search time, another interesting research line involves studying the actual impact of compression in terms of energy within a compact block-addressing inverted index.

Acknowledgments

This work was partially funded by MCIU/FEDER-UE BIZDEVOPS-GLOBAL: RTI2018-098309-B-C32; by MINECO/FEDER GINSENG-UCLM (TIN2015-70259-C2-1-R); and is also part of the SOS project (SBPLY/17/180501/000364), Regional Government of the Autonomous Region of Castilla – La Mancha.

The group from A Coruña is funded in part by EU H2020 RISE BIRDS grant [No 690941]; by Xunta de Galicia/FEDER-UE [CSI: ED431G/01 and GRC: ED431C 2017/58]; by Xunta de Galicia Conecta-Peme 2018 [Gema: IN852A 2018/14]; by MINECO-AEI/ FEDER-UE [ETOME-RDFD3: TIN2015-69951-R; Datos 4.0: TIN2016-78011-C4-1-R; BIZDEVOPS: RTI2018-098309-B-C32].

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, Text compression. Prentice-Hall, Inc., 1990.

[2] I. H. Witten, A. Moffat, and T. C. Bell, Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, 1999.

[3] A. Fariña et al., "Word-based self-indexes for natural language text," ACM Transactions on Information Systems (TOIS), vol. 30, no. 1, p. 1, 2012.

[4] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval, 2 ed. Addison-Wesley Publisihing Company, 2011.

[5] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," ACM Transactions on Information Systems (TOIS), vol. 18, no. 2, pp. 113-139, 2000.

[6] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá, "Lightweight natural language text compression," Information Retrieval, vol. 10, no. 1, pp. 1-33, 2007.

[7] E. Kern et al., "Sustainable software products—Towards assessment criteria for resource and energy efficiency," Future Generation Computer Systems, vol. 86, pp. 199-210, 2018.

[8] G. Pinto and F. Castor, "Energy efficiency: a new concern for application software developers," Communications of the ACM, vol. 60, no. 12, pp. 68-75, 2017.

[9] J. Mancebo et al., "EET: a device to support the measurement of software consumption," in Proceedings of the 6th International Workshop on Green and Sustainable Software (GREENS'18) ACM, 2018, pp. 16-22.

[10] D. A. Huffman, "A method for the construction of minimum-redundancy codes," Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, 1952.

[11] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no. 10, pp. 762-772, 1977.

[12] R. N. Horspool, "Practical fast searching in strings," Software: Practice and Experience, vol. 10, no. 6, pp. 501-506, 1980.

[13] C. Calero and M. Piattini, "Puzzling out software sustainability," Sustainable Computing: Informatics and Systems, vol. 16, pp. 117-124, 2017.

[14] Calgary Corpus. Available: http://www.data-compression.info/Corpora/CalgaryCorpus/. Last access: 19/09/2019

[15] TREC. Available: https://trec.nist.gov. Last access: 19/09/2019

[16] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá, "New adaptive compressors for natural language text," Software: Practice and Experience, vol. 38, no. 13, pp. 1429-1450, 2008.

[17] C. Wohlin et al., Experimentation in software engineering. Springer Science & Business Media, 2012.

[18] J. Mancebo et al., "Assessing the Sustainability of Software Products - A Method Comparison," presented at the EnviroInfo, Kassel (Germany), 2019.

[19] J. S. Culpepper and A. Moffat, "Enhanced byte codes with restricted prefix properties," in Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE'05) Springer, 2005, pp. 1-12.

 ISBN: 978-1-61208-751-1