Onto how Compression yields Energy-efficient text search[1]

Antonio Fariña[a], Olivia Poy[b*], Nieves R. Brisaboa[a], Coral Calero[b], Mª Ángeles Moraga[b], Óscar Pedreira[a]

[a]Centro de Investigación CITIC, Databases Lab., Universidade da Coruña, A Coruña, Spain.
[a]Institute of Technology and Information Systems, University of Castilla-La Mancha, Ciudad Real, Spain.

[1]An early partial version of this article appeared in *Proc GREEN'19* [12].

*Corresponding author(s). E-mail(s): Oscar.Poy@uclm.es;
Contributing authors. E-mail(s): antonio.farina@udc.es; brisaboa@udc.es; Coral.Calero@uclm.es; MariaAngeles.Moraga@uclm.es; oscar.pedreira@udc.es;

Abstract

In the last two decades, word-based text compression has shown to be the key to efficiently handling large collections of text not only due to yielding important storage savings but, more importantly, because it allowed boosting the performance of some traditional text retrieval systems. The reason is that when the appropriate compression techniques are chosen, compressed text search becomes much faster than searches on plain text, and retrieval/decompression could start at any part of the compressed data, hence allowing to keep the text collection compressed all the time. Word-based text compressors have been typically compared in terms of their compression effectiveness, encoding/decoding speed, and performance when searching for words. In this paper, we show that compression also has benefits in terms of energy efficiency when performing word-based searches. Particularly, our experiments considering searches performed over uncompressed text and text compressed with the most well-suited compressors for text databases showed energy savings of around 30-70%. These savings are obtained thanks to improvements in search time but also, rather unexpectedly, because compressed text searches typically require less power from the processor. We have also analyzed how small modifications to the Horspool search algorithm can lead to time savings and can reduce even further the power needs of the processor (up to 5-10%), and, consequently, the overall energy consumption.

Keywords: Text compression, Dense Codes, Huffman Codes, Text Search, Energy efficiency, Green Software

Statements and declarations

Conflict of interest

The authors certify that they have no affiliation with any organization or entity with financial or non-financial interest in the subject matter or materials discussed in this manuscript.

Acknowledgements

This work was partially funded by MCIN/AEI/10.13039/501100011033 and EU/ERDF: grant PID2021-122554OB-C33 (OASSIS); by MCIN/AEI/ 10.13039/501100011033 and European Union NextGenerationEU/PRTR: grant TED2021-129245B-C21 (PLAGEMIS).

The group from Castilla - La Mancha is funded in part by CECD (JCCM) and FEDER funds: grant SBPLY/21/180501/000115 (EMMA).

The group from A Coruña is also funded in part by MCIN/AEI/10.13039/501100011033 and "NextGenerationEU"/PRTR: grants PDC2021-121239-C31 (FLATCITY-POC), PDC2021-120917-C21 (SIGTRANS), and PID2020-114635RB-I00 (EXTRA-Compact); by GAIN/Xunta de Galicia: grant GRC: ED431C 2021/53; by UE, (ERDF), GAIN, Convocatoria Conecta COVID: grant IN852D 2021/3 (CO3).

1 Introduction

Keeping text compressed and having the ability to search for any text pattern within it without performing a prior decompression is a long-standing problem referred to as the *compressed pattern matching problem* [2]. It takes special interest when one aims at keeping a potentially large amount of data compressed to save space but still needs to search within it. Due to its interest, many works [3, 20, 29, 35, 52, 55, 56] aimed at providing efficient compressed text capabilities on top of traditional compressors such as Huffman [27] or Lempel-Ziv [74, 75].

In this scenario, the target was typically focused on supporting searches without decompressing the data, even at the cost of yielding worse performance than when those searches were performed over an uncompressed version of those data, or even slower than simply pipelining the decompression of the data with searches on the uncompressed data [54–56]. In this case, the searches on the uncompressed text/data could be performed with the fastest *string matching algorithms* [53] among which we could find those from the Boyer-Moore family [8] which permit to skip processing some data, or the BNDM strategy based on a suffix automaton exploiting bit-parallelism [51].

Nevertheless, the development of mid-size text retrieval systems such as Glimpse [43] raised the interest of efficiently supporting compressed text searches. Glimpse partitioned a collection of text-documents into blocks of fixed size, and used a two-stage search procedure that initially filtered out the blocks that contained the terms from a given pattern (using a small index requiring only 2-4% of the text size), and then sequentially searched for the queried pattern within those (rather large amount of) candidate blocks. In practice, Glimpse showed that, by developing ad-hoc compression formats supporting fast searches without the need for previous decompression such as BPE [42], not only space savings were achieved (rather poor compression ratios, around 50-60% of the original size in the case of BPE), but the reduced I/O operations could also lead to an important performance boost at query time, making some searches faster than searching the uncompressed text.

Similarly, in the late nineties, the boom of the creation of document repositories, digital libraries, and the grown of the Web, brought the development of full-text retrieval systems for large document collections [69], where a huge inverted index keeping references to the actual positions of words, permitted to efficiently locate any word or phrase within the collection. To save space, both the inverted lists [5, 16, 48, 57, 68, 76] and the text could be compressed [36, 49, 69, 70], in such a way that the whole compressed text database (using *Huffword*, word-oriented binary Huffman [69]) plus the index would occupy around 60-70% of the size of the uncompressed text collection and decompression became only needed when a relevant document had to be delivered to an user. To further save space, document-oriented (or parameterizable block-oriented) inverted indexes could be used instead [6, 69]. Now, the posting list for each word only points to either the documents or blocks where such word occurs. Similarly to Glimpse, this index permits to filter out the documents/blocks where a word occurs at query time. Yet, to gather the actual positions within them, further scanning of the candidate documents/blocks is mandatory. In addition, to solve phrase-pattern queries, the index is only capable of identifying candidate documents/blocks but scanning is again required to verify if the terms in the searched pattern actually occurred at adjacent positions. The development of new byte-aligned compression schemes [11, 50] slightly sacrificing compression effectiveness yet supporting very efficient Boyer-Moore type compressed pattern matching (up to 8 times faster than searches over plain text), permitted to boost the efficiency of those document/block-addressing inverted indexes. Indeed, using just 35-45% of the space required by the source text collection, it became possible to handle both the index and the compressed text while supporting fast queries taking a few milliseconds per query.

In this paper, we consider compression for text databases from a different point of view. Despite considering compressors according to their effectiveness, the ability to perform random decompression (accessing any point and decompressing from there on), compression times, and performance both at decompression and search time, which are traditionally the most important features, we also consider their energy consumption.

Energy consumption is the amount of energy needed by the hardware to run a software. How the software was developed has a direct impact on how it runs and the energy consumption it requires. The development of energy-efficient software, also known as Green software, has recently become a hot concern that tackles this problem [31, 61].

Although an isolated action of a software could need just a small amount of energy and, as such, would emit a low amount of CO_2, when we consider the massive amount of software running in the world at any given time, the perception changes. For example, in the case of data, [15] estimates that 463 EB of new data will be created every day by 2025, while [18] indicates that the digital data universe will grow from 97 trillion GB in 2022 to 181 trillion GB in 2025. All these data need to be stored and managed and therefore most of the current and future created data is stored on data centers which will require among 1137 and 7933 Twh by 2030 according to [4].

The relevance of Green software is emphasized in various works [14, 19, 58, 61] urging software developers to aim at minimizing energy consumption without compromising the functionality of the software, as well as considering its data energy consumption implications.

Since the energy consumption of a computer highly depends on the power used (by its components, e.g. CPU, DRAM, GPU, etc.) and on the execution time, reducing the time needed to run a given process is probably the most straightforward way to decrease its energy consumption. Yet, we must also consider the impact of the power demands. Several works [1, 39, 59, 60, 62, 66, 72] have investigated this effect, showing that reducing time does not always reduce energy consumption. It is therefore necessary to consider more factors. Aspects such as the compiler/code optimizations performed [1], the programming language used [39, 60], the data structures considered [59], and the balance between power needs and the number of threads in multi-threaded applications [62, 66], among others, can have an important effect in terms of energy efficiency.

In the context of data-related considerations, several studies have focused on reducing the energy consumption in data use. For example, [41] and [63] study the impact of database query optimizations on energy consumption. In [41] the authors achieve an almost 500 times reduction in query energy consumption by applying techniques like indexes, covered queries, and ordered queries. Similarly, in [63] the authors achieve a reduction in energy consumption of up to 25% by applying two green software practices, namely "put application to sleep" and "use efficient queries". Following this line, [71] presents a model for estimating the energy consumption of relational queries with an estimation error of less than 10%. This model proved to be an effective tool for the energy optimization of database queries.

Several studies have relied on compression as a way to reduce the amount of data that need to be handled and decrease energy needs. In [73], significant improvements in energy efficiency were achieved for the encryption/transmission of large quantities of data by combining Huffman and MedSecrecy algorithms. In [40], they present a data compression strategy that allows to compress near end data while adapting to the efficiency requirements of green edge computing.

In the scope of large text databases, as indicated above, compression permits to reduce both the size of the inverted indexes and the text itself. A study [22] comparing several integer representations commonly used to represent posting lists within inverted indexes concluded that compact integer vectors are a very suitable alternative in terms of energy-efficiency. To the best of our knowledge, there is not any work focused on studying the energy consumption when performing searches over a large uncompressed or compressed text, and/or when recovering/decompressing text data.

In this paper we present an experimental study mainly focused on performing searches with the fast string matching Horspool algorithm [26] (from Boyer-Moore family) both over uncompressed text, and text compressed with either *End-Tagged Dense Code (ETDC)* [11] or *Tagged Huffman (TH)* [50], some of the best compression techniques for text databases according to [6]. We also tested *Plain Huffman (PH)*, a slightly more effective compressor, where searches follow a different approach that makes them less time and energy efficient in practice.

We complement our study with a comparison where we include typical metrics regarding compression and decompression processes in `ETDC`, `TH`, and `PH`, and we also evaluate in terms of energy efficiency.

We have run experiments using two different energy-measurement tools: an Energy Efficiency Tester (EET) device [44] providing real measurements, and the Linux `perf` tool. Our energy measurements permitted us to verify that, when using Horspool-based searches, compressed pattern matching is both faster and more energy efficient than searches over plain text. Our comparison with searches performed over `PH`, following non-Horspool-based approach permitted us to conclude that there are several variables such as the number of instructions executed, the number of branches and branch-misspredictions, cache-friendliness, that matter in terms of energy efficiency. This led us to try different small modifications (tricks) over the original Horspool algorithm, and we observed that they could yield additional savings in energy and/or time.

In the following section, we revise word-based compressors, the most-suited techniques for text databases, paying special attention to those supporting Horspool-based searches (`ETDC` and `TH`). Then, in Section 3 we focus on searches. We present the Horspool string matching algorithm, and show several simple modifications that permit to improve its performance (and energy requirements) in practice. In Section 4, we present the framework used for measuring energy consumption. Finally, we present our experiments in Section 5, and provide our conclusions in Section 6.

2 Word-based compression

We present the basics of semistatic word-based compression and pay special interest to the three word-based compressors that will be used in the experimental section; i.e. `TH`, `PH`, and `ETDC`. We not only review their general compression and decompression procedures, but also give details regarding their particular encoding schemas. This is of particular interest when we compare them empirically.

2.1 Basic concepts on word-based text compression

Classic compression techniques such as Huffman [27] or Lempel-Ziv [74, 75] are rather unattractive for text databases. In the case of Lempel-Ziv, even though they offer reasonable compression capabilities, and offer the

possibility of performing searches within the compressed data [55, 56], as other adaptive compressors, they do not support direct access within the compressed text; i.e. accessing a given part and recovering data from there on. Supporting direct access involve the need for using semistatic compressors, where each symbol in the source text is replaced by a unique codeword in the compressed data. The classic Huffman compressor is the most-representative example of semistatic compressor. Being character-oriented, it obtains the number of occurrences of each character in the text, and following Huffman algorithm, assigns each character a prefix-free codeword in such a way that more frequent symbols receive shorter codewords. Obtaining prefix-free codewords is of interest, in terms of performance, because it permits to univocally decode any codeword without the need for checking any lookahead. Unfortunately, by being character-oriented, classic Huffman yielded very poor compression (compression ratio around 60% in English texts). Also, by generating bit-oriented codewords (i.e. codewords are a sequence of bits), its performance at decompression and search, was rather slow in practice.

The raise of word-based compression [7, 46] becomes a key break-through in this field. By using words instead of characters as the source symbols of the compressors, compression was largely improved. Not only there were much less codewords in the compressed data, but also the frequency distribution became much more biased than when using characters, which was of interest for statistical compression. The result is that, a word-based compressor such as Huffword [69], which used bit-oriented Huffman to give each word a codeword, reached strong compression ratios, close to 25%, on English texts. However, due to using bit-oriented codewords, the performance at decompression and searching was still rather poor.

Plain (`PH`) and Tagged Huffman (`TH`) [50] tried to solve those limitations. They were also word-oriented compressors and used Huffman for encoding. However, they used byte-oriented (D-ary) rather than bit-oriented Huffman. Therefore, the codewords assigned to each word are now sequences of bytes. Dealing with bytes at decompression clearly improved decompression speed (bit-manipulations were avoided) at the cost of a loss in compression ratio. In the case of `PH`, compression ratios were around 30%. As in classic Huffman, `PH` could support direct access to the compressed text, but only from the codeword beginnings. That is, `PH` is a non self-synchronizable code; i.e. we cannot access any byte in the compressed text, move a few bytes to the left or right in order to detect the beginning of a codeword, and continue decompression from there on. To provide self-synchronism, `TH` follows the same approach as `PH`, but reserves the first bit of each byte in a codeword to mark if that byte is the first byte (bit set to 1) or not (bit set to 0). Therefore, while `PH` uses 8-ary Huffman (i.e. it uses the 2^8 bit combinations in each byte), `TH` uses 7-ary Huffman on the remaining 7-bits of each byte. Reserving the flag bit to mark the beginning of each codeword worsened compression ratios to around 35%, but brought self-synchronization capabilities and made codewords suffix-free. The later property means that a codeword cannot appear as a suffix of a longer codeword, and permitted `TH` to perform searches within the compressed text with the fast string matching algorithms from the Boyer-Moore family[8], which boosted its performance. In practice, searches for both single words or sequences of words (phrases) become up to 8 times faster than searching plain text [50]. Its performance at searches, fast decompression, random access, and self-synchronization capabilities pushed `TH` as the best choice to compress text databases.

In 2003, End-Tagged Dense Code (`ETDC`) was presented. It followed the same ideas of `TH`, yet using the flag bit to mark the end of the codewords instead of marking the beginning. This small change made codewords gain the prefix-free property without the need for using Huffman algorithm, which allowed `ETDC` to use the whole 2^7 bit-combinations of each byte. In consequence, `ETDC` obtained compression ratios around 31%; i.e. much better compression than `TH` and almost reaching the compression ratios of `PH`. Furthermore, by reserving a flag-bit in each byte, it retained the same self-synchronizing and search capabilities of `TH`. This boosted `ETDC` as the most suitable compressor for text databases [6].

2.2 Two-pass compression: general structure

All the above compressors follow a word-based semistatic (or two-pass) approach. They process the source text twice at compression time. The compression process is illustrated in Figure 1.

During the first pass, the compressor creates a model of the text being processed that is composed of the different words/symbols occurring in the text (i.e. the vocabulary of symbols[1]), and their number of occurrences. Then, it sorts the vocabulary by frequency. Next, the according to statistical compression, the compressor performs an encoding stage where the most frequent words receive shorted codewords, hence gaining compression. The particular encoding schemas of `TH` and `PH`, as well as `ETDC` will be discussed below (see Sections 2.4 and 2.3).

Then, the compressor traverses again the text (second pass), and replaces each word/symbol from the source text by its corresponding codeword. A *compressed file* storing all those codewords is created. In addition, to allow decompression, a *header* is also stored along with the compressed data. The header keeps the correspondence symbol↔codeword, which basically consists of the vocabulary words sorted by frequency in `ETDC`, plus a few integers (to save the Huffman-tree shape) in the case of Huffman-based techniques.

[1] In word-based compression, where the symbols are words, the compressor typically arranges words into a hash table to efficiently handling the vocabulary. In particular, this enables fast checking if a word already belongs to the vocabulary and also increasing its frequency count each time a word is input.

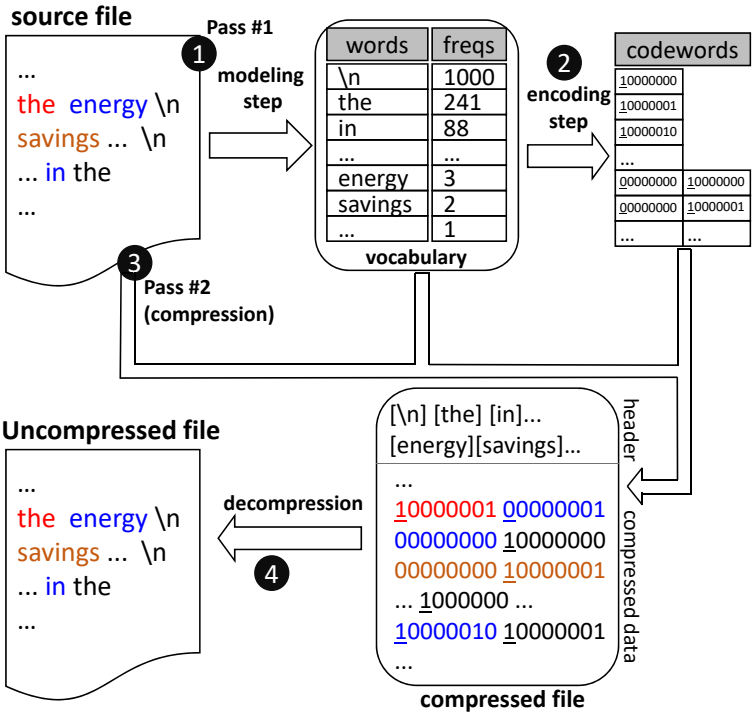

Fig. 1: Structure of compression and decompression processes in two-pass compression.

Note that, since a unique codeword replaces each source symbol/word, we can directly perform searches within the compressed text. Given a word being searched for, we must obtain the corresponding codeword, and then traverse the compressed file to gather the positions where that codeword occurs.

Regarding decompression, the process is still simpler: the decompressor loads the header file into memory, arranging an array of words V. Then, the decompression process traverses the compressed file, decoding a codeword c_i at a time. Decoding c_i returns an integer i corresponding to the position in the vocabulary of the associated word $w_i = V[i]$, which is output to the file that will keep the recovered uncompressed data.

2.3 Encoding and Decoding procedures in PH and TH

As indicated above, both PH and TH use Huffman algorithm [27] to ensure the generation of prefix-free codewords, yet they are byte-oriented. In a binary Huffman, once all the symbols (words) are sorted by frequency, Huffman algorithm assigns each symbol (word) a leaf node in a Huffman tree which is built by taking iteratively the two least frequent nodes of the tree which are set as children of a new internal parent node (whose frequency is set as the sum of the frequency of its two children). This process is repeated until only a root node remains to be paired. After that, we assume that left-children are labelled with a 0, and right-children with a 1, and the symbols, represented as leaves, are assigned a codeword which consists in the path from the root down to the corresponding leaf. For decoding, each bit b_i from a codeword is used to traverse the Huffman tree from the root to the leaves. If b_i is a 0 we move to the left child of the current node (starting at the root), and otherwise we move to the right child. When a leaf is reached, the decoding of a codeword ends and the corresponding symbol is reported.

The same applies to build a D-ary (byte-oriented) Huffman code [50] in PH and TH. However, by being byte-oriented, the process of building the Huffman tree takes the $R = 2^8 = 256$ least-frequency nodes in each iteration.[2] In TH, $R = 2^7 = 128$ byte-values are allowed instead due to the reserved flag bit within each byte.

In practice, the construction of a D-ary Huffman tree can be done in place [47], and by using canonical Huffman codes [65], both the encoding and decoding processes can be simplified. Basically, a canonical Huffman code retains the same number of codewords of each length (i.e. leaves at each level of the tree), but ensures that, in each level of the Huffman tree, leaves are set in the left-most part and internal nodes are in the right-most part. This simple change, ensures that all codewords in each level of the tree are given consecutive numerical values, and permits to implicitly keep the Huffman-tree shape by using just a couple of tables $first$ and $base$ that respectively store, for each possible codeword length $1 \leq l \leq L$: $base[l]$ the rank in the vocabulary V of the first symbol receiving a codeword of length l; and $first[l]$ the numerical value of the first codeword of length l (i.e. the codeword of the first leaf at level l), or $first[l] = M = (2^D)^l$ if no codeword of length l exists.

[2] Only for the first iteration, the number of children of the first (parent) internal node created is set as $2 \leq R \leq 2^8$, to ensure that all the internal nodes have 2^8 children [21, 50]. From there on, $R = 2^8$ is set.

Figure 2 gives an example showing how a binary ($D = 1$) Huffman code is built for a given text. It also includes the corresponding canonical Huffman code, and the final codeword assignment for each source symbol. We also include the tables $first$ and $base$. Algorithm 1 shows the pseudocode for decoding a canonical Huffman code, using those tables, following the ideas in [25]. This is the decoding algorithm used in our experiments for both PH ($D = 8$) and TH ($D = 7$).

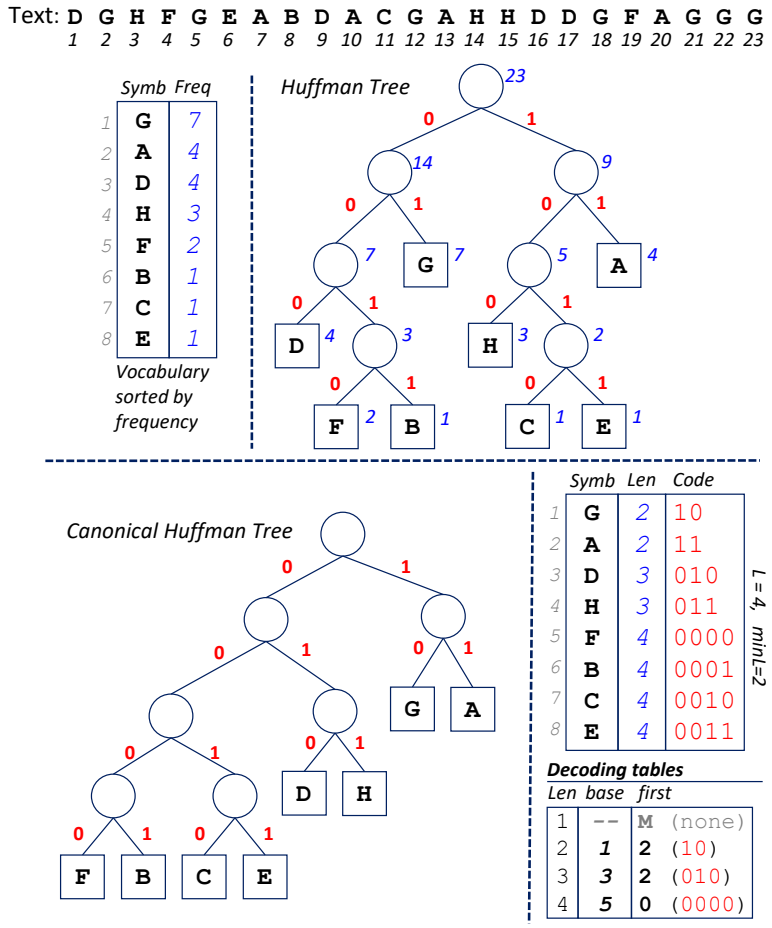

Fig. 2: Construction of a binary Huffman tree (top). Corresponding canonical Huffman tree (botom-left). Final codeword assignment and decoding tables (bottom-right).

Algorithm 1 Decoding D-ary canonical Huffman. `Canonical-Decode(`C`)`

Input: A codeword C of length $l \leq$ L
Input: Arrays $base[1..L]$ and $first[1..L]$
Output: The rank of the decoded symbol within V

1: l=1; j=C[l]
2: **while** j < first[l] **do**
3: j= j * 2^D
4: $l = l$ +1
5: j= j + C[l]
6: **end while**
7: **return** base[l] + j - first[l] ;

2.4 Encoding and Decoding procedures in ETDC

ETDC owns a simple encoding schema. As in TH, it reserves the first bit of each byte in a codeword, but now marks with 1 the last byte of a codeword rather than the first byte. Therefore, the last byte of each codeword is of the form 1xxxxxxx (numerical value ≥ 128), and the preceding bytes (in codewords of ≥ 2 bytes) have numerical values smaller than 128 (i.e. 0xxxxxxx).

The consequence of marking the end of the codewords is that ETDC gains the prefix-free property without the need for using Huffman (as in PH and TH), and ETDC can use the whole 2^7 combinations of the remaining 7 bits in each byte to create the codewords. Therefore, once the vocabulary is sorted by frequency, ETDC performs a sequential codeword assignment to each word, and the codeword given to each word does no longer depend on its actual number of occurrences (as in Huffman-based counterparts), but only on its rank within the sorted vocabulary. Those codewords are assigned as follows:

- 1-byte codewords of the form 1xxxxxxxx (byte values from 128 to 255), are given to the first 128 words in the vocabulary.
- Next 128^2 words are given 2-byte codewords of the form 0xxxxxxx:1xxxxxxx; i.e. codewords from 0:128 to 127:255.
- Next 128^3 words are given 3-byte codewords of the form 0xxxxxxx:0xxxxxxx:1xxxxxxx; i.e. from 0:0:128 to 127:127:255.
- If needed, 4-byte codewords would be assigned to the next 128^4 words, and so on.

Even though codewords are sequentially assigned to words during compression, the simple dense encoding allows also simple on-the-fly encoding and decoding algorithms [11]. The decoding algorithm, used in our experiments, is included in Algorithm 2. It uses a lookup table $base[1..5]$ indicating the rank of the first word encoded with $1, 2, \ldots, 5$-byte codewords within the vocabulary V; i.e. $base \leftarrow [1, 1+128, 1+128+128^2, 1+128+128^2+128^3, 1+128+128^2+128^3+128^4]$.

Algorithm 2 Decoding End-Tagged Dense Codes. ETDC-Decode(C)

Input: A codeword C of length $l \leq$ L
Input: Array $base[1..L]$
Output: The rank of the decoded symbol within V

```
1: j=0;   l=1
2: while C[l] < 128  do
3:     j= j * 128 + C[l]
4:     l = l +1
5: end while
6: j= j * 128 + (C[l] -128)
7: return base[l] + j;
```

3 Performing searches

In the next section, we present the Horspool algorithm, a fast string matching algorithm that was also shown to be the best choice to perform searches within text compressed with TH and ETDC. We discuss the original Horspool algorithm, and also introduce a small modification that is mandatory to allow applying it over text compressed with ETDC. Then, we also present some small variations (tricks) that can be used to improve its performance in practice. Finally, we also discuss a simple non-Horspool-based algorithm to perform searches within PH.

3.1 The Boyer-Moore type Horspool Algorithm

Pattern-matching algorithms gather the positions within a text $T[1..n]$ where a given pattern (a sequence of symbols) $P[1..m]$ occurs. Both T and P are composed of symbols drawn from a given alphabet Σ. A brute-force algorithm takes $O(mn)$ time and is very inefficient in practice. The Horspool algorithm [26] has also worst-case complexity of $O(mn)$, but it is sublinear in practice [53] because, during the sequential traversal of T, some parts can be skipped. Horspool uses a search window of m symbols (see Figure 3), which is moved forward over T. At each step, the pattern is compared (right-to-left) with the symbols in the window. If they match, a new occurrence of the pattern is reported. Otherwise, no match occurs. In any case, the sliding window is moved forward as many positions as possible, yet guarantying no matches are lost.

In practice, for each possible symbol β, the longest safe shift ($d[\beta]$) can be precomputed using Algorithm 3. Basically, if β is the last symbol in the window and j is the position of the last occurrence of β within P, the longest safe shift is $m - j$. If β only occurs at $P[m]$, or β does not occur in P, the skip distance is set to m.

The complete pseudocode for Horspool [53] is included at Algorithm 4. Note that the sliding window is always set at positions $T[i+1..i+m]$, and each iteration of the while loop at Line 3 corresponds to one

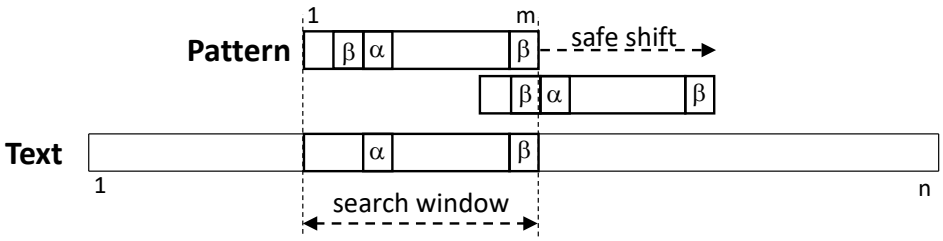

Fig. 3: Horspool search algorithm scheme.

alignment of the sliding window with P. Lines $5-6$ perform the right-to-left comparison of P with the window. If they match, Line 9 outputs the position of the match (i.e. $T[i+1]$).[3]

The efficiency of Horspool is associated to its ability to perform small jumps forward along of the text (Line 11), ensuring no matches are lost. In general, the longest safe jump depends on the length of the pattern (m), and having a large alphabet decreases the probability of match between P and T. Therefore, large values of m and $|\Sigma|$ would raise the chances to skip processing more text, and consequently, lead to more efficient searches in practice.

Algorithm 3 Preprocessing step in Horspool. `Pre-Process(`P`,`d`)`

Input: A search pattern P built on alphabet Σ
Result: d, safe shift for each symbol

```
1: for s ∈ Σ do
2:     d[s] = m
3: end for
4: for i=1 to m-1 do
5:     d[p[i]] = m-i
6: end for
```

Algorithm 4 Horspool search algorithm. `Horspool-Search(`T`,`P`)`

Input: Text $T[1\,..\,n]$ built on alphabet Σ
Input: Search pattern $P[1\,..\,m]$ built on Σ
Output: Positions where P occurs at T

```
1: Pre-process(P, d)
2: i=0;   end=n-m
3: while i ≤ end do
4:     j= m
5:     while j > 0 and P[j]=T[i+j] do
6:         j= j-1
7:     end while
8:     if j=0 then
9:         output i+1
10:    end if
11:    i= i + d[T[i+m]]
12: end while
```

[3] Alternatively, if we were interested in counting the number of matches rather than outputting the actual matching positions, Line 9 could be replaced by increasing a counter of matches (i.e. matches = matches+1).

3.1.1 Using Horspool within compressed text

Although Horspool is typically run over plain text, it can be used to perform direct searches within text compressed with `TH` [50] or `ETDC` [11, 50]. Since in those compressors each codeword in the compressed text is associated to a unique word from the vocabulary of the compressor, given a word w being searched for, we first obtain the codeword C_w associated to w, and then search for the pattern $P = C_w$ using Horspool.

Therefore, the searcher program runs an initializing step which consists in loading all the words from the vocabulary into a hash table, and associating them their corresponding codeword. This takes $O(V)$ time, being V the number of different words. After that we can perform searches for any word within the compressed text, and retrieving the codeword C_w for any word w takes $O(1)$ time.[4]

In the case of `TH`, we can use the original Horspool algorithm illustrated in Algorithm 4. This is possible because, `TH` generates *suffix-free* codewords, i.e. a short codeword is never a suffix of other codeword [50]. This property guarantees that when Horspool finds a match (at Line 9), it will be an actual match of the codeword being searched for.

However, using the original Horspool to search for a given pattern $P = C_w$ over text compressed with `ETDC` would lead to reporting false matches. In this case, the problem is that the codewords generated by `ETDC` are not *suffix free*. Therefore, for example, if we are looking for the 2-byte codeword `0:128`, we could find occurrences of the form `x:0:128` within the compressed text. Note that, being x the byte preceding each occurrence of the searched pattern, if $x < 128$ we have found a false occurrence of `0:128`, because it is just the suffix of a longer 3-byte codeword `x:0:128`. Otherwise, if $x \geq 128$ an actual match is reported, because x is the last byte of a previous codeword. This property gives a simple way to discard false matches during the search, yet implies that a slight modification has to be applied to Horspool algorithm [11]. Basically, we must replace Lines $8-9$ from Algorithm 4 (`if j=0 output i+1`) to include the additional check; i.e. `if j=0 and (i=0 or T[i] >=128) then output i+1`. Note that this does not worsen search performance in practice as the check is only done upon the detection of a match, and the number of matches is much lower than the size of the text (n).

3.2 Tweaking Horspool

Based on our implementation in [11], we can consider some tricks to try to improve the efficiency of Horspool algorithm. We illustrate them in Algorithm 5:

Algorithm 5 Tweaked Horspool search algorithm. `Horspool-Search-trick(`T`,`P`)`

Input: Text $T[1\,..\,n]$ built on alphabet Σ
Input: Search pattern $P[1\,..\,m]$ built on Σ
Output: Positions where P occurs at T

```
1:  Pre-process(P, d)
2:  i=0;   end=n-m
3:  while true do
4:      while P[m]≠T[i+m] do
5:          i= i + d[T[i+m]]
6:      end while
7:      j= m-1
8:      while j > 0 and P[j]=T[i+j] do
9:          j= j-1
10:     end while
11:     if j=0 then
12:         if i+1 > n then
13:             return
14:         else
15:             output i+1
16:         end if
17:     end if
18:     i= i + d[T[i+m]]
19: end while
```

- *Skip-check-end.* We can append P at the end of T to avoid the need for checking whether the end of T was reached within the main loop (`while i<= end`). Such condition is now only checked when a match is found (see Line 9), which may barely occur. Therefore, Line 3 becomes now a `while(true)` loop and the search stops, at Line 13, when we reach the occurrence of P that was appended to T.

[4] We could also use an array of words sorted lexicographically, and perform binary search to fetch any word in $O(\log V)$ time.

- *Skip-loop.* The inner loop from Algorithm 4 (see Line 5) has a double condition: checking if `j>0` and if `T[i+j]=P[j]`. If the probability of matching between the last symbol of P ($P[m]$) and the last symbol of the search window ($T[i+m]$) is low, we can add a *skip-loop* that moves the search window forward while $T[i+m] \neq P[m]$, and only when they are equal, starts the right-to-left check. Note that now the double condition is only needed in the right-to-left loop, which actually starts with `j=m-1`. Lines $4-7$ from Algorithm 5 include this modification.

The tricks above aim at reducing the number of comparisons done during the search process, which is an usual metric to compare search algorithms [38]. An additional language-dependent trick we can exploit when using `C`, is using pointer-based syntax for arrays. Algorithm 6 provides a `C`-based implementation of Algorithm 5 where accesses to T were replaced by pointers `cpos` and `cj`. This has the potential to reduce the number of instructions involved in the accesses to array T.

Algorithm 6 Tweaked C-style Pointer-based Horspool. `Horspool-Search-C` (T, P)

Input: Text $T[1..n]$ built on alphabet Σ
Input: Search pattern $P[1..m]$ built on Σ
Output: Positions where P occurs at T

```
1:  Pre-process(P, d)
2:  cpos=&T[m]
3:  cend=&T[n-1]
4:  while true do
5:      while P[m] ≠ *cpos  do
6:          cpos= cpos + d[*cpos]
7:      end while
8:      j= m-1; cj= cpos-1
9:      while j > 0 and (P[j] == *cj) do
10:         j= j-1; cj = cj -1
11:     end while
12:     if j=0 then
13:         if cpos ≥ cend then
14:             return
15:         else
16:             output cj+1
17:         end if
18:     end if
19:     cpos= cpos + d[*cpos]
20: end while
```

Unless stated, the Horspool-based searchers used in our experiments will follow the schema presented in Algorithm 6, which was the choice used in [11]. Yet, to reduce memory usage, the text is read from disk buffer-wise using a 1 MiB-buffer.

In addition, in Section 5.3, we will compare this buffer-wise variant with other different implementations that do not use buffered-input (the whole file is read at once) and differ in which "tricks" are enabled.

3.3 Searching text compressed with `PH`

A simple and flexible way to search for word-based patterns within text compressed, was presented in [50] and applied to `PH`. The idea is very simple. Given a word w, we obtain its position i within the vocabulary, and mark such position with a flag. Then, the search process simulates the decompression of the compressed data, sequentially decoding each codeword c_j. If j is the rank in the vocabulary corresponding to c_j, the search algorithm simply compares if $i = j$. In such case, a match is reported, otherwise the searcher continues decoding the next codewords. Even though this strategy is simple, it can still exploit the fast decoding performance of byte-wise decoders to yield reasonable search times. However, it cannot reach the speed of the fast Horspool-based counterparts in practice [11].

4 Measuring energy consumption

We have used two different approaches to measure the energy consumption when performing our experiments. The first one is based on a hardware device that captures real consumption of the computer where each experiment is run. The second one uses an estimator, a software tool that estimates the energy consumption of the computer during a time period. We decided to use both approaches to: (1) compare the difference between

real measurements and estimations; and (2) validate that the results obtained were correct. We provide details of both approaches below.

4.1 The FEETINGS framework

The framework FEETINGS (Framework for Energy Efficiency Testing to Improve eNvironmental Goals of the Software) [45] helps to measure and analyze the energy consumed by a software product that runs in a computer. FEETINGS comprises three components: (i) a conceptual component known as GSMO (Green Software Measurement Ontology), which standardizes the terminology used in the study, (ii) a methodological component known as GSMP (Green Software Measurement Process) that ensures measurement consistency and reliability, and (iii) a technological component, which in turn consists of two artifacts:

- An Energy Efficiency Tester (EET) [44] which is a hardware tool that measures and records the energy consumed by the computer and some of its components when a software is run on it.
- The ELLIOT tool, which processes and analyses the data obtained by EET, providing the results to the user in a user friendly way.

On FEETINGS, the computer where the software to be measured is run is called Device Under Test (`DUT`). In this study, our `DUT` has the following characteristics: an Intel i7 10700k processor, 2 modules of 16 GB of RAM (DDR4@ 3200Mhz), a 500 GB SSD Western Digital Blue drive, GPU ATI sapphire radeon X1950 GT and a 3GO PS580S 580w power supply with 1 fan. It runs Ubuntu 20.04.2 LTS and the compiler used was gcc version 6.2.0.

4.2 Using RAPL: The Perf tool

Intel's Running Average Power Limit (RAPL) [28] was introduced in the Sandy Bridge architecture [17]. Since then, modern intel processors include so-call Model Specialized registers (MSR) that can be read to obtain an estimation of the energy consumption of different processor domains. Among them, we can find: (i) `energy-pkg`, indicating the energy wasted by the whole processor package which includes all the cores in the processor and the memory controller, the last level cache, among others (ii) `energy-cores`, which involves the energy used by all the cores; and (iii) `energy-ram`, which draws RAM energy values. Yet, it is available from Haswell processors on.

Many studies [23, 24, 30, 32–34] have focused on RAPL as a way to estimate energy usage, or included methods to evaluate its accuracy. In practice, since the performance counters are periodically updated (roughly around once per millisecond), estimations are particularly accurate when we measure long-term energy consumption. By gathering the values at the beginning and end of the execution of a given program, it is possible to have a good estimation of the energy usage in Joules ($w \cdot s$, or watt second), and to also obtain the average power usage (in watts) as avg_power = energy/execution_time.

There exists different APIs/tools that permit to access the values provided by the RAPL interface. For example, the `PAPI` project [13, 67], `Pinpoint` tool [37], or the Linux tools `turbostat`, `likwid-perfctr`, and `perf`. Among them, we have chosen `perf` (the performance analysis tools for Linux) to gather different metrics in our experiments. Apart from the energy counters indicated above (`energy-pkg`, `energy-cores`, and `energy-ram`), `perf` allows also access to many other processor performance counters (see `perf` *list*). Among them, we can see values (events) associated to the number of `cpu-cycles`, `cpu-instructions`, `cpu-branch-instructions`, `cpu-branch-misses`, `L1-data-cache-loads`, `L1-data-cache-misses`, `LLC-cache-loads`, and `LLC-cache-misses` associated to the processor during a period of time. From them we can also derive the number of cycles required per instruction (`CPI`) as `CPI` = `cpu-cycles`/ `cpu-instructions`.

In addition, to ease running experiments `perf` *stat -a [-r 100] -e event1 -e event2* `<program args>` permits to track those events along the executions of a given program (using *-r* parameter allows to repeat the program $n = 100$ times and finally shows average values).

We also used the tool `likwid-perfctr`, which similarly to `perf`, also supports access to all the performance counters we needed. However, in this case this tool simplified the access to the instantaneous values of the counters at regular periods, which is interesting to track their evolution along time. More details are included in Section 5.3.1.

5 Experimental Results

We run experiments with the main goal of evaluating both the performance and the energy consumption of text searches performed over plain text and compressed text searches performed directly over text compressed with `ETDC`, `TH`, and `PH`.

We used a large text from [11] which is composed of the concatenation of the small *CALGARY* corpus[5], and all the texts included in the seven collections from TREC-2 and TREC-4;[6] i.e. AP Newswire 1988, Ziff Data 1989-1990, Congressional Record 1993, and Financial Times from 1991 to 1994. The size of the resulting text is around 1.03 GiB and it has has 886,190 different words out of 228,707,250 total words when parsed using the spaceless-word model [50].

We have used `ETDC`, `TH`, and `PH` to compress our dataset, and present results showing typical statistics such as compression ratio[7] and both compression and decompression times. These results are presented in Section 5.1. Then, in Section 5.2 we pay our focus on searches. Apart from showing search times to evaluate performance, we also provide some additional measurements obtained with FEETINGS framework (mainly processor energy and overall energy in the DUT) as well as with Linux `perf` tool, including both the energy-related and the performance metrics discussed in Section 4.2. Finally, in Section 5.3, we focus on searches over text compressed with `ETDC`, and evaluate several variants of the original Horspool algorithm in terms of energy and performance.

We have used two test computers. Our main computer has an intel i7-8700k@3.7GHz processor, 64Gb RAM (DDR4@2400Mhz) RAM, and a 1Tb nvme unit. It runs Ubuntu 20.04 (kernel Linux 5.4.0-137-generic), and the compiler used is gcc 9.4.0. Unless specified, all the measurements shown below are referred to this machine. The second computer is the DUT from FEETINGS framework. We will refer to its measurements as DUT-values.

5.1 Compressing text with `ETDC`, `TH`, and `PH`

We have compressed our dataset with three word-based text compressors: `ETDC`, `TH`, and `PH`. In all cases, our compressors generate a file that keeps the list of vocabulary words (and some extra integers in `TH` and `PH` to store the shape of the Huffman tree), plus a second (large) file with the encoded data.

5.1.1 Space requirements

As expected, `PH` obtains slightly better compression ratio (32.83%) than `ETDC` (33.55%), and `TH` yields clearly the worst compression ratio (36.38%). Results are presented in Table 1. We also show that, on average, each symbol (word) is encoded with 1.51, 1.55, and 1.68 bytes respectively in `PH`, `ETDC`, and `TH`.

Table 1: Compression: Space results.

Metric	ETDC	TH	PH
encoded-data size (KiB)	346,978	375,681	338,260
vocabulary size (KiB)	8,261	8,261	8,261
total size (KiB)	355,239	383,941	346,521
compression ratio (%)	33.66	36.38	32.83
Avg code length (bps)	1.55	1.68	1.51

5.1.2 Compression: performance and other metrics

At compression, the three techniques perform exactly the same two-pass process which was discussed in Section 2.2. That is, a first pass over the source dataset to gather the different words and compute their relative frequencies is followed by a sorting of the vocabulary by frequency, and by a code-generation phase which is slightly simpler in `ETDC` than in `PH` and `TH`, and where a codeword is associated to each different word. Finally, a second pass over the text replaces each word by the assigned codeword to make up the final compressed file, and all the vocabulary words are saved into the vocabulary file. In consequence, the results drawn by the three techniques are roughly the same. In Table 2, we present measurements including compression (and decompression) times as well as energy and performance metrics.

The fact that `TH` compresses less than the others, and that `ETDC` has a simpler encoding procedure, bring some slight differences. For example, `ETDC` and `PH` are around 1% faster at compression, and despite having a slightly slower encoding, `PH` is slightly faster than `ETDC` due to its better compression (less data must be output). In any case, the fact that Huffman encoding uses a couple of small tables during the code generation phase (and also at decoding), leads to both `TH` and `PH` make around 5% more L1-cache accesses than `ETDC`, and allows `ETDC` to shorten the gap with `PH`.

Considering energy metrics and CPU performance counters, the three techniques draw almost the same values in terms of `power-cores` and `power-pkg`, with `TH` again requiring slightly more processor power than the others. Curiously, probably due to its larger average codeword length, `TH` shows a 4% increased value in `cpu-branch-misses`, and the increased number of branch miss-predictions is a probable source of energy

[5] http://www.data-compression.info/Corpora/CalgaryCorpus/
[6] https://trec.nist.gov
[7] The compression ratio (in percentage) is computed as *compressed_text_size/source_text_size* ×100.

Table 2: Compression and Decompression: Energy and performance metrics. Average values from 10 runs.

Metric	Compression			Decompression		
	ETDC	TH	PH	ETDC	TH	PH
user time (sec)	1.45	1.47	1.45	0.52	0.55	0.53
elapsed time (sec)	1.48	1.49	1.47	0.54	0.57	0.56
energy-pkg (J)	55.93	56.35	55.61	20.90	22.20	21.58
energy-cores(J)	39.27	39.61	39.03	14.77	15.70	15.24
energy-ram(J)	3.30	3.32	3.29	1.17	1.23	1.20
power-pkg(watt)	37.81	37.92	37.78	38.56	38.67	38.53
power-cores(watt)	26.54	26.66	26.51	27.25	27.35	27.22
power-ram(watt)	2.23	2.23	2.23	2.15	2.14	2.14
#cpu-instructions ($\times 10^9$)	10.68	10.77	10.73	3.91	3.97	3.94
#cycles-per-instruction (CPI)	0.69	0.68	0.68	0.67	0.7	0.68
#cpu-branch-instructions ($\times 10^9$)	1.84	1.84	1.84	0.63	0.65	0.64
branch-misses (%)	3.15	3.28	3.16	5.33	5.44	5.2
L1-dcache-loads ($\times 10^9$)	2.99	3.18	3.17	1.11	1.31	1.25
LLC-loads ($\times 10^9$)	0.04	0.04	0.04	0.01	0.01	0.01

waste in the processor [64] due to the need to undo all the speculative work done. Finally, since the energy (in Joules=$w \cdot s$) wasted by any component can be seen as its instantaneous power (in watts) multiplied by the elapsed time (in seconds), we can see how again `PH` uses less energy than `ETDC`, and the later less than `TH`.

Finally, we can also observe that, as a traversal observation applying to all the experiments we performed, the `power-ram` values are roughly the same for the three techniques.

5.1.3 Decompression: performance and other metrics

The decompression process starts by loading the vocabulary words into memory, and then traversing the compressed file. For each codeword, a decode algorithm recovers the position in the vocabulary of the word associated to that code, which is fetched and written to the output file. The unique difference here is that the Huffman-based techniques must access two 4-integer arrays for each byte processed, whereas `ETDC` does only make one access to a 4-integer array [11] when a whole codeword is decoded (recall arrays *base* and *first* in Algorithms 1 and 2). Consequently, `ETDC` has much less `L1-data-cache-loads`. In addition, due to the simple decoding algorithm in `ETDC`, the number of instructions executed is lower than that of the Huffman-based counterparts and the number of `cycles-per-instruction` is slightly better. The result is that, despite decompressing a larger file than `PH`, `ETDC` is the fastest technique at decompression and consequently also wastes slightly less energy than the others.

Comparing the `power-ram` parameter between compression and decompression, we can see that decompression requires less power for the RAM. This is probably because in compression, access to RAM does not follow a completely sequential-access pattern due to the need for searching for each word from the source text within the vocabulary (which is handled with a large hash table). Instead, the vocabulary of the decompressor is simply an array of words where the most frequent words appear within the same region. This improves the locality, and is noticed by the fact that the number of `LLC-cache-loads` is much smaller at decompression. Of course, as compression takes roughly three times more time than decompression, it also requires around three times more processor energy (values `energy-cores` and `energy-pkg`). Finally, it is also interesting to see that the processor uses more power (`power-pkg` and `power-cores`) at decompression than at compression. In our opinion, the simple decoding algorithms (a loop with at most 4 iterations), is probably allowing the processor to perform speculative execution of instructions. Yet, the larger value of `cpu-branch-misses` indicates that the percentage of branch miss-predictions is around 60-70% higher than during compression. The additional work implied by the need of undoing the instruction associated to those miss-predictions could be the source of the increase of processor power at decompression.

5.2 Experiments on searches

In this section, we compare the results obtained when performing searches for single-word patterns with Horspool algorithm over plain text, or over text compressed with `ETDC` or `TH`. We also included in the comparison searches performed over text compressed with `PH`. Recall that for `PH` we cannot use Horspool algorithm, and we simulate the decompression process instead, which is typically slower than the Horspool-based counterparts. Our search programs are configured to read input using 1 MiB input buffer, and both *skip-check-end* and *skip-loop* tricks were enabled. Recall those tuning options were discussed in Section 3.2. In addition, instead of reporting the actual matching positions of the patterns, our searchers will only count the total number of matches.

5.2.1 Generating single-word pattern sets

Aiming at searching for single-word patterns both over compressed and plain text, we have generated four different sets of patterns, composed of 100 words each, that vary both in number of occurrences and word length. To create them we followed two steps: First, we considered all the 886,190 words that compose the vocabulary of out text collection and followed the model in [50], that assumes that all words are searched for with uniform probability, and we randomly extracted words of varying length. In particular, we obtained two sets containing respectively words of 4 characters (ch4) or more than 8 characters (ch8+).

Recall that the longest jumps available for Horspool algorithm depend on the length of the patterns being searched for, and that, for plain-text searches, this is exactly the number of characters of the words. Besides, for compressed searches, pattern length is referred to as the codeword length, and consequently it depends on the words frequency. Therefore, in a second step, we considered the frequency/number-of-occurrences of the words to filter those two sets (ch4 and ch8+) by frequency. In particular, we extracted from them words occurring between 2,776 and 203,130 times to create pattern sets c2-ch4 and c2-ch8+, and words occurring between 23 and 611 times to create pattern sets c3-ch4 and c3-ch8+.

The reason why we chose those frequency ranges is to ensure that the number of bytes of the codewords given by both `ETDC` and `TH` to all the words in sets c2-ch4 and c2-ch8+ is exactly two. To accomplish this, we verified that, in `ETDC`, the words ranked from 128 to 16511 (i.e. those being assigned two-byte codewords) in the vocabulary decreasingly-sorted by frequency have a number of occurrences between 203,131 and 612. Similarly, in `TH` words ranked from 75 to 6,023, which are assigned two-byte codewords, have frequency between 348,994 and 2,776. Indeed, the word ranked at position 6,024 has frequency 2,775. Therefore, by choosing words whose frequency ranges from 2,776 to 203,130, we ensure they are ranked within positions 129 and 6,023, and consequently, both `ETDC` and `TH` assign them two-byte codewords.

We proceed analogously to create pattern sets c3-ch4 and c3-ch8+ so that all the words in those sets are assigned three-byte codewords both by `ETDC` and `TH`. Note that in `TH`, words receiving three-byte codewords are ranked from position 6,024 to 106,818 in the vocabulary and their frequency ranges from 2,775 to 22. Besides, in `ETDC`, all the words ranked from position 16,512 on are assigned three-byte codewords, and the word ranked at position 16,512 has frequency 612. Therefore, words within c3-ch4 and c3-ch8+ must have frequency between 23 and 611 to ensure they are ranked at positions within the range $[16512, 106818]$.

The main statistics of these query sets are summarized in Table 3. In columns 2 and 3 we show the codeword length in bytes and the average number of characters of the words for each query set. Then, in columns 4 and 5 we show respectively the average number of occurrences found in the case of compressed text searches (column 4) and those found for plain text searches (column 5). Note that they differ because in compressed-text searches we look for actual occurrences of the codewords, whereas in the case of plain-text searches, results include not only actual occurrences of a pattern w but also matches of the string w occurring within longer words. Note that, for long patterns, the number of occurrences is rather similar because there are not usually longer words including the search patterns. However, for short patterns the number of matches in plain-text searches is up to around $3-22$ times higher than the actual number of occurrences.

Table 3: Query sets used and their main statistics.

Query set name	codeword length (bytes)	Word length (chars)	Average matches found	
			codewords (#occs)	words (#occs)
c2-ch4	2	4.0	17,892	55,337
c2-ch8+	2	9.3	8,947	10,509
c3-ch4	3	4.0	101	2,276
c3-ch8+	3	9.9	116	160

5.2.2 Performing searches

Table 4 includes some results involving both search time and energy metrics obtained in our two test computers: our main computer (the one with the i7-8700k CPU, we refer to it as `PC-1` below), and the `DUT` from FEETINGS. We show both the execution (elapsed) time and the energy measurements. The later were obtained as the value of `energy-pkg` provided by `perf` in `PC-1`, whereas for `DUT`, those measurements correspond to the actual `cpu-energy` values obtained through the FEETINGS framework. We complement those results with Figure 4.

Note that we also have included cpu-power values (in watt) in Table 4, which are computed dividing the total amount of energy (J) by the elapsed running time (sec).

Yet, in practice, the real measurements obtained for `DUT` draw values that are 26-29% higher than those provided by `perf` in `PC-1`. Despite those gaps, this result is well-aligned with the achievements of previous works [33, 34] where RAPL measurements showed to be highly correlated with the actual energy values. In any

Table 4: Searches for single-word patterns in our two test computers: `perf` values for `PC-1`, and actual values for `DUT`. Showing Energy (J) and elapsed search time (sec.). Statistics reflect average values from 10 runs, each run involving 100 searches for the patterns in each query set.

		PC-1 (perf)			DUT (EET)		
	tool	Elapsed time (sec)	Energy pkg (J)	Power pkg (watt)	Elapsed time (sec)	Energy pkg (J)	Power pkg (watt)
c2-ch4	ETDC	51.36	1661.03	32.34	51.90	2145.40	41.33
	TH	50.91	1651.32	32.44	51.53	2140.56	41.54
	PH	81.70	3046.14	37.28	81.08	3954.69	48.78
	Plain	120.55	4211.32	34.94	117.57	5276.40	44.88
c2-ch8+	ETDC	51.21	1662.83	32.47	51.53	2153.55	41.79
	TH	50.88	1649.51	32.42	51.57	2103.27	40.78
	PH	81.75	3051.69	37.33	80.95	3958.10	48.90
	Plain	78.28	2777.72	35.48	77.42	3499.64	45.21
c3-ch4	ETDC	35.77	1169.41	32.69	36.75	1494.80	40.68
	TH	35.17	1150.34	32.71	36.49	1470.36	40.30
	PH	81.77	3054.27	37.35	81.07	3939.38	48.60
	Plain	111.79	3840.39	34.35	109.88	4867.21	44.30
c3-ch8+	ETDC	35.65	1160.11	32.54	36.50	1505.79	41.25
	TH	35.20	1154.41	32.80	36.44	1461.46	40.10
	PH	81.80	3052.52	37.32	80.98	3952.52	48.81
	Plain	76.23	2706.65	35.50	75.19	3424.13	45.54

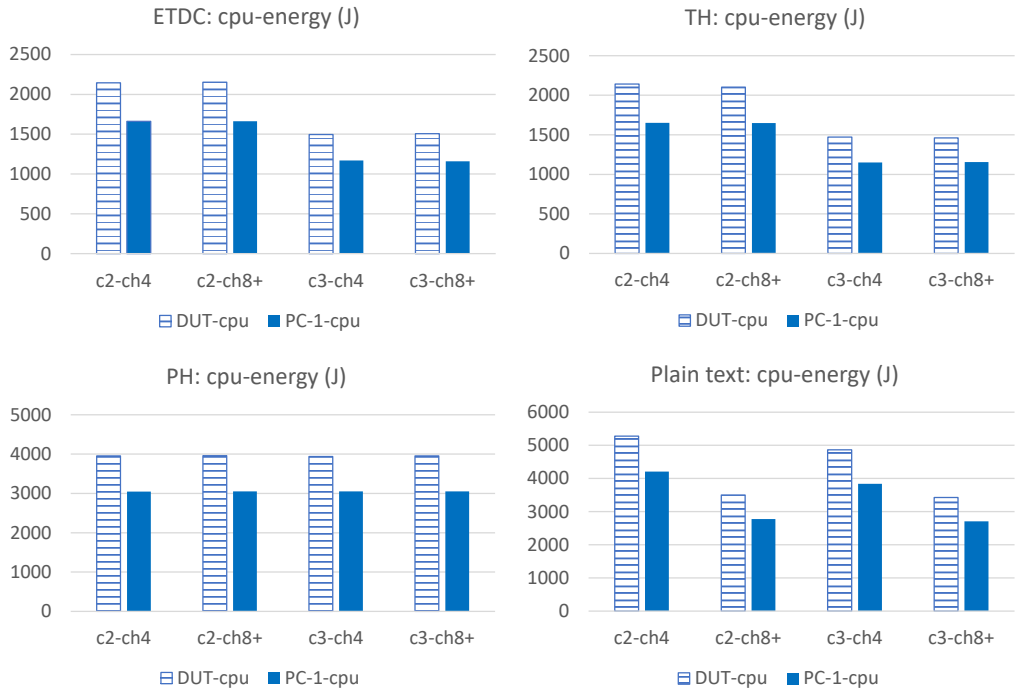

Fig. 4: Energy comparison between our two test computers: `perf` values for `PC-1`, and actual values for `DUT`. Showing average energy (in J) from 10 runs, each run involving 100 searches for the patterns in each query set.

case, we could expect a slightly higher consumption from the processor in `DUT` in comparison with that of `PC-1` from the fact that both processors use 14 nm lithography, but the TDP[8] of the i7-8700k processor (`PC-1`) is 95 watt and it reaches 4.7GHz, whereas the processor i7-10700k (`DUT`) has a higher TDP (125 watt) and reaches 5.1GHz.

Entering into details, we can see that the search times from our experiments largely vary depending on the query set. Searches with Horspool algorithm within compressed text are (as expected [11]) the fastest alternative overall. We can see that `ETDC` and `TH` obtain almost identically search times (with `TH` being slightly faster), and they are clearly faster than `PLAIN` (i.e. Horspool over plain text). In practice, we can observe that, even when `TH` must search for small 2-byte codewords (pattern sets c2-ch4 and c2-ch8+), the elapsed time obtained by `PLAIN` is around $1.5-2.3$ times worse than that of `TH`. That gap increases to $2.1-3.3$ times slower when `TH` searches for 3-byte patterns (pattern sets c3-ch4 and c3-ch8+).

[8]TDP from Thermal Design Power.

Some considerations must be taken into account. Despite the space of TH and ETDC being around 1/3 than that of PLAIN, recall that from Table 3 we can see that, for pattern-set c2-ch8+, the words searched for in PLAIN are (in average) 3.3 times longer than the compressed patterns searched for in TH and ETDC. In practice, the longest forward-jump for PLAIN is close to 10 bytes, while in ETDC and TH is of just 3 bytes. This would benefit PLAIN. On the contrary, the number of different byte values within compressed text in text compressed with ETDC, TH, and PH is 256, while in plain text, only 163 different characters appeared (mainly characters [0-9],[a-z],[A-Z], space, line_break, and punctuation symbols). According to [9], this leads to the fact that the probability of a match between the data and the pattern being searched is much higher in plain text (0.05) than in compressed text (0.01 in ETDC and TH), which increases the number of comparisons, between the text and the pattern in PLAIN. This can be seen in Table 5 where we show extended metrics for the same searches, which include access to the performance counters gathered by perf tool. We can observe that both the number of cpu-branch-instructions and the percentage of cpu-branch-misses is around $2-3$ times higher in PLAIN than in TH. Rather unexpectedly, our results show that TH is slightly faster than ETDC at searches. Indeed, in [11] ETDC was always faster than TH due to is better compression ratio. In our opinion, the lower percentage of cpu-branch-misses in TH is the only metric that can explain why ETDC is slightly slower in practice. Recall that ETDC and TH search algorithms differ only in the fact that, after each match, ETDC has an extra if to check if it is an actual valid match of the compressed pattern (see Section 3.1.1).

Regarding the performance of the searches over text compressed with PH, we can see that they do not depend on the type of the patterns, because they always simulate the decompression of the compressed text, and after decoding each codeword, check if the decoded position matches the position of the word being searched for in the vocabulary. In practice, searches over PH are clearly slower than Horspool-based searches over TH and ETDC. They are even slower than searches with Horspool over plain text, except when short words are dealt with.

Focusing on the energy consumption of the processor, the main achievement is rather expected: those searches requiring less elapsed time are also more energy-efficient. Considering DUT, a look at Table 4 and Figure 4 shows that 100 searches in TH and ETDC roughly require 2100J and 1500J respectively for high and low frequency patterns. Instead, in PLAIN in the best/worst scenario, those 100 searches roughly require from 2700J to 5300J. This indicates that searches with Horspool over compressed text required $29-55\%$ less energy than over plain text. Regarding searches on PH, we can see that the cost of 100 searches is always around 4000J, i.e. it uses around $1.9-2.6$ times more energy than Horspool over TH and ETDC. Similar achievements are obtained in our computer PC-1, as well as when we look at the total energy values gathered from the PSU.

If we finer analyze our results, and particularly when we look at power measurements, we can see that the power used by the processor (also by ram), slightly varies depending on the search algorithm run. Consequently, the *energy* (obtained as $power \times time$) does not only depend on the elapsed time, but also on the power used by the components along time. Below we analyze the results included in Table 5.

- ETDC vs TH. Both Horspool-based searchers draw almost exactly the same values for energy-pkg, energy-cores, and energy-ram. The smaller percentage of cpu-branch-misses in TH seems to be compensated by the need for processing more data. The later implies that TH has to execute more cpu-instructions per second than ETDC (note the similar user-times, and a smaller value for CPI in TH) and also requires a slightly higher number of L1-data-cache-loads and LLC-cache-loads.
- PLAIN vs Horspool over compressed text (TH and ETDC). We can clearly see that Horspool-based searches over compressed text (TH and ETDC) require around $8-10\%$ less power from the processor than searches over plain text (PLAIN). In this case, the $1.5-4$ times smaller percentage of cpu-branch-misses gives advantage to the compressed text searches. It is also interesting to see that the around 3 times larger amount of data being processed, not only leads to a 3-times higher value of LLC-cache-loads, but also increases the power needed for accessing the main memory. Consequently, values for power-ram increase from around 2.2 watt (in TH and ETDC) to around $2.6-2.8$ watt in PLAIN, which implies around $18-27\%$ more power needs.
- Searches with PH. An initial comparison with the values of energy-pkg and energy-ram for PH when considering searches (Table 5) and decompression (Table 2), permits us to verify that those values roughly match; i.e. they are around 37.3 watt and 2.1 watt respectively in both cases. This is expected as the search algorithm simulates the decompression process. When comparing power values for the processor, we can see that PH requires more power (see power-pkg and power-cores values) than the Horspool-based searchers. In practice, it needs around $13-23\%$ more processor power than TH and ETDC, and even around $5-12\%$ more power than PLAIN. This is not surprising due to the high percentage of cpu-branch-misses (around 14%), and a particularly high value of L1-data-cache-loads due to the access to the decoding tables of Huffman.

5.3 Regular and tweaked variants of Horspool algorithm

In this section, we focus on performing searches over text compressed with ETDC, considering different Horspool variants that basically differ in which tweak-options are enabled (or not). Those options were discussed in Section 3.2: i.e. (1) reading input-data using either a *small* 1-MiB buffer or a *large* buffer where the whole

Table 5: Searches for single-word patterns in computer `PC-1`. Showing energy and performance metrics. Statistics reflect average values from 10 runs, each run involving 100 searches for the patterns in each query set.

		Time (sec)		Energy (J)			Power (watt)			#cpu	CPI	#cpu	brnch	#L1	#LLC
	tool	user	elapsed	cpu pkg	cpu cores	ram	cpu pkg	cpu cores	ram	inst x10^9		brnch x10^9	miss (%)	loads x10^9	loads x10^9
c2-ch4	ETDC	47.90	51.36	1661.03	1079.86	110.36	32.34	21.02	2.15	120.72	2.09	20.76	0.84	39.44	0.51
	TH	47.48	50.91	1651.32	1075.24	109.73	32.44	21.12	2.16	130.00	1.93	22.26	0.68	42.60	0.54
	PH	77.98	81.70	3046.14	2121.78	172.77	37.28	25.97	2.11	479.18	0.84	73.45	14.72	178.11	0.56
	Plain	87.67	120.55	4211.32	2851.47	313.59	34.94	23.65	2.60	313.21	1.90	55.33	3.10	94.78	1.54
c2-ch8+	ETDC	47.74	51.21	1662.83	1083.40	110.05	32.47	21.16	2.15	121.61	2.07	20.90	0.86	39.69	0.51
	TH	47.26	50.88	1649.51	1073.70	109.76	32.42	21.10	2.16	129.85	1.93	22.23	0.67	42.59	0.55
	PH	77.71	81.75	3051.69	2126.78	172.85	37.33	26.02	2.11	478.66	0.84	73.41	14.73	178.03	0.56
	Plain	45.30	78.28	2777.72	1892.23	223.09	35.48	24.17	2.85	207.12	1.87	36.63	2.76	61.16	1.48
c3-ch4	ETDC	32.51	35.77	1169.41	764.67	78.28	32.69	21.38	2.19	82.70	2.12	14.29	0.92	27.05	0.49
	TH	31.58	35.17	1150.34	752.36	77.30	32.71	21.39	2.20	87.53	1.97	15.03	0.65	28.62	0.52
	PH	77.62	81.77	3054.27	2129.11	172.99	37.35	26.04	2.12	480.24	0.84	73.69	14.68	178.34	0.56
	Plain	78.91	111.79	3840.39	2578.93	294.80	34.35	23.07	2.64	291.62	1.89	50.47	1.54	88.58	1.53
c3-ch8+	ETDC	32.17	35.65	1160.11	756.71	78.02	32.54	21.22	2.19	82.69	2.12	14.29	0.86	26.90	0.49
	TH	31.69	35.20	1154.41	756.09	77.56	32.80	21.48	2.20	91.72	1.91	15.78	0.69	29.63	0.52
	PH	77.70	81.80	3052.52	2127.08	173.04	37.32	26.00	2.12	480.42	0.84	73.69	14.67	178.52	0.56
	Plain	43.04	76.23	2706.65	1844.09	218.89	35.50	24.19	2.87	203.18	1.86	35.83	2.60	59.55	1.49

#cpu-inst= `#executed cpu-instructions`; CPI = `#cycles-per-instruction`;
#cpu-brnch= `#executed cpu-branch-instructions`; #brnch-miss= `percentage of cpu-branch-misses`;

data is read with only one system-call; (2) using either an *array*-based or a *pointer*-based implementation; (3) either using a *skip-loop* or not; and (4) either using *skip-check-end* trick or not. Results are included in Table 6. We provide six variants that are labeled $v_{0..6}$.

The First variant (v_0) corresponds exactly to the original Horspool illustrated in Algorithm 4. In this case, the whole text is kept into memory (buffer = *large*), and no tricks were enabled. We can compare it with counterpart v_1 which only differs in having **skip-check-end** trick enabled. Therefore, it is exactly as Algorithm 5, yet replacing Lines $5-7$ by just `j=m` (without *skip-loop*). When comparing both versions, we obtain almost exactly the same search times. However, in v_1, the processor must execute around 23% less `cpu-instructions`, 33% less `cpu-branch-instructions`, and almost half `L1-data-cache-loads`, than in v_0. Despite a 50% increase of the number of `cpu-branch-misses`, which also increases the `CPI` value by around 30%, the result is that the cpu uses around 5% less power (`power-pkg`) than in v_0.

If we also enable the **skip-loop** trick, we get version v_2. In this case, as expected, both the number of `cpu-instructions` and `cpu-branch-instructions` decreases even further than in v_1, being respectively around 19% and 50% lower than in v_1, and consequently around 35% and 65% lower than those values in v_0. Thanks to that, the power required by the processor also decreases, with values of `power-pkg` around 12% lower than those of v_0. Surprisingly, the elapsed time is around $8-10$% worse in v_2 than in v_0 because the average `CPI` increases by around 70% as the result of a huge increase of the number of `cpu-branch-misses` which is around 3 times higher in v_2. The additional work associated to those miss-predictions would also increase the number of `L1-data-cache-loads`. Unfortunately, the decrease of `power-pkg` values in v_2 is counteracted by the increase in execution time, and both v_0 and v_2 require roughly the same energy at searches, showing that while the *skip-check-end* trick improves energy-consumption, using also the *skip-loop* trick voids such improvement in practice.

While variants $v_{0..2}$ used an array-based implementation, the remaining Horspool variants follow a pointer-based approach to access the data, as illustrated in Algorithm 6. Indeed, v_3 and v_4 are respectively the pointer-based versions of v_2 and v_3. We can see that the pointer-based implementations (slightly) improve searching time as well as the values of power and energy required by the processor with respect to the array-based counterparts. In practice, v_4 (*skip-check-end* and *skip-loop* enabled) improves almost all the values of the metrics drawn by v_2. Comparing v_3 and v_1 (only *skip-check-end* is enabled), the pointer-based strategy v_3 obtains slightly better times, processor power, and energy-usage values. However, rather surprisingly, v_3 executes around $8-9$% more instructions than v_1. This seems to be compensated by an around 10% lower `CPI` value.

Finally, we can compare v_4 and v_5; i.e. we enable both *skip-loop* and *skip-check-end* options and vary how input data is read. Recall in v_4 the whole text is loaded into memory with a unique `read()` system call, whereas in v_5 we issue a `read()` call to load the next 1MiB of data. This reduces the memory requirements of the searcher program, yet worsens almost all the rest of the metrics involved. The only exception is *user time* which is slightly better in v_5. Of course, *elapsed time* clearly worsens because now the operating system must attend a higher number of system-calls during the life-time of the searcher process.

To sum up, we have observed that, increasing the size of the input-buffer to reduce the number of system calls involved in our Horspool variants improves both performance and energy consumption. We could also observe how small modifications in the source code, can yield additional profit. We observed that even though coupling *skip-check-end* and *skip-loop* tricks (v_2) permitted us to slightly improve the energy requirements of the original Horspool algorithm (v_0), surprisingly at the cost of slightly worsening search-times, the Horspool variant (v_1), which enables *skip-check-end* but disables *skip-loop* trick, yields the best overall results.

5.3.1 Measurements with likwid

We have used `likwid-perfctr` software to obtain instantaneous values for energy metrics along the execution of our Horspool-based search alternatives for text compressed with `ETDC`. We ran the tools as: `likwid-perfctr -t 100ms -C S0:1 -g ENERGY -o out.xml -O <searcher args>`; i.e. the searching program is run over core 1 from socket 0 in our processor. We obtain metrics of group "ENERGY"[9] every 100 milliseconds, and they are output to file *out.xml*. Figure 5 includes the results. In all cases, the first part of the execution has large variations due to the initialization of the searcher, which includes the load of the vocabulary words into memory, their insertion into a hash table, and assigning them their corresponding codeword. From there on, each search is run. Even though there are slight differences with the average values reported by `perf` in the previous section, we could reach similar conclusions regarding the comparison among the different search variants. That is, the variants with *skip-check-end* and *skip-loop* enabled v_4 and v_2 obtain the lower values for `power-pkg` energy consumption. They are followed by v_3 and v_1 (only *skip-check-end* is enabled), with v_1 being on a par with v_5. Finally, the original Horspool v_0 clearly requires more power from the processor.

[9] Available groups for each type of processor were available at `/usr/local/share/likwid/perfgroups/<type-cpu>/`

Table 6: Searches over `ETDC` for single-word patterns in computer `PC-1`. Different Horspool-based variants are compared. We show energy and performance metrics. Statistics reflect average values from 10 runs, each run involving 100 searches for the patterns in each query set.

	Variant	Buffer	array/ptr	SkipLoop	SkipChckEnd	Time (sec)		Energy (J)			Power (watt)			#cpu	CPI	#cpu	brnch	#L1	#LLC
						user	elapsed	cpu pkg	cpu cores	ram	cpu pkg	cpu cores	ram	inst x10^9		brnch x10^9	miss (%)	loads x10^9	loads x10^9
c2-ch4	v0	l	a			44.83	45.09	1552.35	1045.90	100.81	34.43	23.20	2.24	235.15	0.92	54.22	0.26	125.81	0.05
	v1	l	a		✓	44.73	45.05	1475.10	969.01	100.77	32.74	21.51	2.24	181.98	1.19	36.44	0.40	72.31	0.06
	v2	l	a	✓	✓	48.39	48.60	1538.03	992.06	108.40	31.65	20.41	2.23	146.22	1.59	18.68	0.77	90.16	0.06
	v3	l	p		✓	44.68	44.92	1451.71	947.14	100.51	32.32	21.09	2.24	199.91	1.08	36.43	0.38	90.15	0.06
	v4	l	p	✓	✓	48.15	48.43	1513.65	969.68	108.35	31.25	20.02	2.24	111.21	2.09	18.82	0.76	36.69	0.05
	v5	s	p	✓	✓	47.90	51.36	1661.03	1079.86	110.36	32.34	21.02	2.15	120.72	2.09	20.76	0.84	39.44	0.51
c3-ch4	v0	l	a			30.27	30.53	1050.72	708.16	69.63	34.42	23.20	2.28	158.18	0.93	36.47	0.28	84.57	0.04
	v1	l	a		✓	30.30	30.57	999.42	656.37	69.74	32.69	21.47	2.28	122.52	1.20	24.53	0.42	48.62	0.04
	v2	l	a	✓	✓	33.18	33.44	1056.76	681.50	75.96	31.60	20.38	2.27	98.65	1.63	12.66	0.85	60.66	0.04
	v3	l	p		✓	30.12	30.38	979.86	639.03	69.31	32.26	21.04	2.28	134.34	1.09	24.50	0.42	60.60	0.04
	v4	l	p	✓	✓	33.09	33.36	1039.59	665.20	75.79	31.16	19.94	2.27	74.98	2.14	12.70	0.84	24.75	0.04
	v5	s	p	✓	✓	32.51	35.77	1169.41	764.67	78.28	32.69	21.38	2.19	82.70	2.12	14.29	0.92	27.05	0.49

#cpu-inst= `#executed cpu-instructions`; CPI = `#cycles-per-instruction`;
#cpu-brnch= `#executed cpu-branch-instructions`; #brnch-miss= `percentage of cpu-branch-misses`;

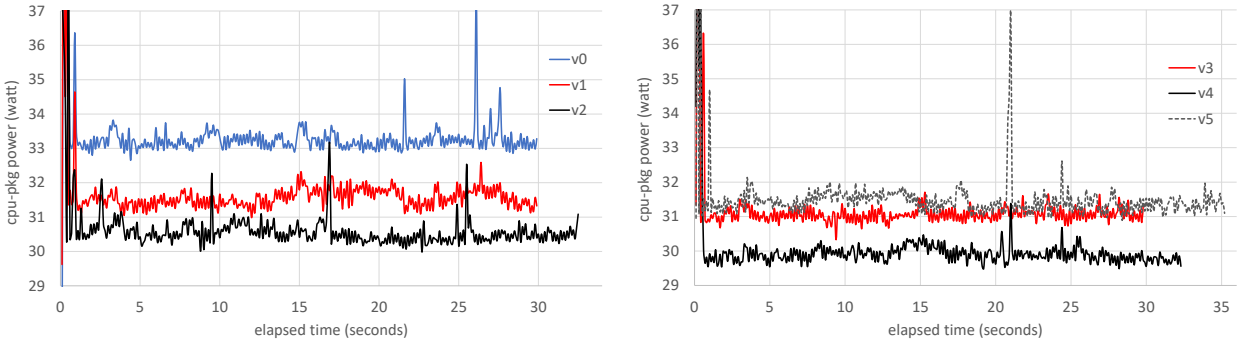

Fig. 5: Evolution of processor `power-pkg` values during one execution of the 100 searches from pattern set c3-ch4, using Horspool-based ETDC search variants. The left/right plots correspond respectively to the array/pointer-based implementations.

6 Conclusions and Future Work

In this paper, we have revisited the problem of compression for text databases. We have reviewed three well-known word-based compressors (`ETDC`, `TH`, and `PH`) specially tailored for the compression of large text collections thanks to their good compression ratio (around 30-35%), the support for random and fast decompression, and the possibility of performing fast searches directly within the compressed text. Those searches are particularly fast within text compressed with `ETDC` and `TH` because the fast Horspool algorithm can be used.

We have compared those compression techniques considering some traditional metrics such as their compression ratio, compression and decompression performance, and search speed. Furthermore, we have tackled the problem of comparing them in terms of the amount of energy they require. To achieve this, we used two different approaches: FEETINGS framework, which includes an energy measuring tool that relies on actual measurements provided by a dedicated hardware device; and the linux `perf` and `likwid-perfctr` tools, which allowed us to obtain power/energy estimations, as well as metrics associated to other internal counters from the processor that can be used to analyze what is happening within the processor during a period of time.

Our experiments allowed us to conclude that, as expected, when comparing the energy consumption of different processes, the running time is the most important metric. However, using `perf`, we could analyze that there are other internal parameters (e.g. cache-usage, number of instructions/branches executed, etc.) that matter in terms of execution time, and more importantly, in terms of energy consumption.

When compressing our text collection with `ETDC`, `TH`, and `PH`, we could observe that, since the three compressors follow almost identical compression procedures, elapsed time directly determines energy consumption. However, at decompression, a fine analysis with `perf` allowed us to understand why, despite processing a larger compressed file, `ETDC` is faster and requires around 5% less energy than `PH` during decompression. In any case, differences in energy consumption are rather small.

We also included experiments where we performed searches for words within both compressed and plain text. On the one hand, we observed that the Horspool-based searches over text compressed with `ETDC` and `TH` are much faster than non-Horspool-based searches within text compressed with `PH`, require around 40-50% less energy, and more interestingly, demand less power from the processor. On the other hand, we compared searches performed using Horspool both over plain text and within text compressed with `ETDC` and `TH`. In general, searches are much faster within compressed text and require roughly 30-50% less energy than over plain text. In addition, thanks to then need for much less memory accesses and a reduced percentage of branch misspredictions, searches over compressed text demand less power from the processor.

Finally, we included some small modifications over the original Horspool algorithm that drawn different performance/energy tradeoffs. Among them, it was rather surprising to observe that using a *skip-loop* trick was able to largely decrease the number of comparisons performed by the original Horspool, as well as the power needs of the processor. But, however, elapsed-times actually worsened and consequently the overall energy-requirements increased.

As future work, we could extend this study to other text compression strategies. In particular, we have created dynamic one-pass word-based compressors for natural language text that are also based on *dense codes* and can be of interest in the scenario of data transmission. Among them, *dynamic lightweight dense codes* [10] obtain compression ratios close to those of `ETDC` and support fast decompression and searches. Finally, since it is now clear that searches within compressed text clearly overcome searches within plain text in terms of performance and energy-efficiency, we consider it could be worth analyzing the overall impact that compression would have in terms of energy when considering a block-addressing inverted index.

References

[1] Abdulsalam S, Zong Z, Gu Q, Qiu M (2015) Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In: Proc. 2015 Sixth International Green and Sustainable Computing Conference (IGSC), pp 1–8, https://doi.org/10.1109/IGCC.2015.7393699

[2] Amir A, Benson C (1992) Efficient two-dimensional compressed matching. In: Proc. Data Compression Conference (DCC), pp 279–288, https://doi.org/10.1109/DCC.1992.227453

[3] Amir A, Benson G, Farach M (1996) Let sleeping files lie: Pattern matching in z-compressed files. Journal of Computer and System Sciences 52(2):299–307. https://doi.org/10.1006/jcss.1996.0023

[4] Andrae A, Edler T (2015) On Global Electricity Usage of Communication Technology: Trends to 2030. Challenges 6:117–157. https://doi.org/10.3390/challe6010117

[5] Anh VN, Moffat A (2005) Index compression using fixed binary codewords. Information Retrieval 8:151–166. https://doi.org/10.5555/1012294.1012301

[6] Baeza-Yates R, Ribeiro-Neto B (2011) Modern Information Retrieval: The Concepts and Technology behind Search, 2nd edn. Addison-Wesley Publishing Company, USA

[7] Bentley J, Sleator D, Tarjan R, Wei V (1986) A locally adaptive data compression scheme. Communications of the ACM 29(4):320–330. https://doi.org/10.1145/5684.5688

[8] Boyer R, Moore J (1977) A fast string searching algorithm. Communications of the ACM 20(10):762–772. https://doi.org/10.1145/359842.359859

[9] Brisaboa N, Fariña A, Navarro G, Paramá J (2008) New adaptive compressors for natural language text. Software: Practice & Experience 38:1429–1450. https://doi.org/10.5555/1455460.1455465

[10] Brisaboa N, Fariña A, Navarro G, Paramá J (2010) Dynamic lightweight text compression. ACM Trans Inf Syst 28(3). https://doi.org/10.1145/1777432.1777433

[11] Brisaboa NR, Fariña A, Navarro G, Paramá JR (2007) Lightweight natural language text compression. Information Retrieval 10(1):1–33. https://doi.org/10.1007/s10791-006-9001-9

[12] Brisaboa NR, Calero C, Fariña A, García F, Mancebo J, Pedreira Ó (2019) Saving energy in text search using compression. In: Proc. 4th International Conference on Green Communications, Computing and Technologies (GREEN). IARIA, pp 1–7, URL https://www.thinkmind.org/index.php?view=article&articleid=green_2019_1_10_80031

[13] Browne S, Dongarra J, Garner N, Ho G, Mucci P (2000) A portable programming interface for performance evaluation on modern processors. Int J High Perform Comput Appl 14(3):189—-204. https://doi.org/10.1177/109434200001400303

[14] Capra E, Francalanci C, Slaughter SA (2012) Is software "green"? Application development environments and energy efficiency in open source applications. Information and Software Technology 54(1):60–71. https://doi.org/10.1016/j.infsof.2011.07.005

[15] Cassidy F (2019) A Day in Data. URL https://www.raconteur.net/infographics/a-day-in-data/

[16] Claude F, Fariña A, Martínez-Prieto MA, Navarro G (2016) Universal indexes for highly repetitive document collections. Information Systems 61:1–23. https://doi.org/10.1016/j.is.2016.04.002

[17] David H, Gorbatov E, Hanebutte UR, Khanna R, Le C (2010) Rapl: Memory power estimation and capping. In: Proc. 16th ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED '10, p 189–194, https://doi.org/10.1145/1840845.1840883

[18] Deering I (2021) The importance of data. URL https://www.raconteur.net/infographics/the-importance-of-data

[19] Dick M, Naumann S (2010) Enhancing sofware engineering processes towads sustainable sofware product design. Envirolnfo 2010: Integration of Environmental Information in Europe pp 706–715

[20] Farach M, Thorup M (1998) String matching in lempel—ziv compressed strings. Algorithmica 20(4):388–404. https://doi.org/10.1007/PL00009202

[21] Fariña A (2005) New compression codes for text databases. PhD thesis, University of A Coruña

[22] Fuentes-Sepúlveda J, Ladra S (2019) Energy consumption in compact integer vectors: A study case. IEEE Access 7:155625–155636. https://doi.org/10.1109/ACCESS.2019.2949655

[23] Hackenberg D, Ilsche T, Schöne R, Molka D, Schmidt M, Nagel WE (2013) Power measurement techniques on standard compute nodes: A quantitative comparison. In: Proc. 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 194–204, https://doi.org/10.1109/ISPASS.2013.6557170

[24] Hackenberg D, Schöne R, Ilsche T, Molka D, Schuchart J, Geyer R (2015) An energy efficiency feature survey of the intel haswell processor. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp 896–904, https://doi.org/10.1109/IPDPSW.2015.70

[25] Hirschberg DS, Lelewer DA (1990) Efficient decoding of prefix codes. Communications of the ACM 33(4):449–459. https://doi.org/10.1145/77556.77566

[26] Horspool RN (1980) Practical fast searching in strings. Software: Practice & Experience 10(6):501–506. https://doi.org/10.1002/spe.4380100608

[27] Huffman DA (1952) A method for the construction of minimum-redundancy codes. Proc IRE 40(9):1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[28] Intel (2022) Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[29] Kärkkäinen J, Navarro G, Ukkonen E (2003) Approximate string matching on ziv-lempel compressed text. Journal of Discrete Algorithms (JDA) 1(3):313–338. https://doi.org/10.1016/S1570-8667(03)00032-7

[30] Kelley J, Stewart C, Tiwari D, Gupta S (2016) Adaptive power profiling for many-core hpc architectures. In: Proc. 2016 IEEE International Conference on Autonomic Computing (ICAC), pp 179–188, https://doi.org/10.1109/ICAC.2016.45

[31] Kern E, Hilty LM, Guldner A, Maksimov YV, Filler A, Gröger J, Naumann S (2018) Sustainable software products—towards assessment criteria for resource and energy efficiency. Future Generation Computer Systems 86:199–210. https://doi.org/10.1016/j.future.2018.02.044

[32] Khan KN, Nybäck F, Ou Z, Nurminen JK, Niemi T, Eulisse G, Elmer P, Abdurachmanov D (2015) Energy profiling using igprof. In: Proc. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp 1115–1118, https://doi.org/10.1109/CCGrid.2015.118

[33] Khan KN, Ou Z, Hirki M, Nurminen JK, Niemi T (2016) How much power does your server consume? estimating wall socket power using rapl measurements. Computer Science - Research and Development 31(3):207–214. https://doi.org/10.1007/s00450-016-0325-4

[34] Khan KN, Hirki M, Niemi T, Nurminen JK, Ou Z (2018) Rapl in action: Experiences in using rapl for power measurements. ACM Trans Model Perform Eval Comput Syst 3(2). https://doi.org/10.1145/3177754

[35] Klein ST, Shapira D (2005) Pattern matching in huffman encoded texts. Information Processing and Management 41(4):829–841. https://doi.org/10.1016/j.ipm.2003.08.008

[36] Klein ST, Bookstein A, Deerwester S (1989) Storing text retrieval systems on cd-rom: Compression and encryption considerations. ACM Transactions on Information Systems 7(3):230–245. https://doi.org/10.1145/65943.65946

[37] Köhler S, Herzog B, Hönig T, Wenzel L, Plauth M, Nolte J, Polze A, Schröder-Preikschat W (2020) Pinpoint the joules: Unifying runtime-support for energy measurements on heterogeneous systems. In: Proc. 2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), pp 31–40, https://doi.org/10.1109/ROSS51935.2020.00009

[38] Lecroq T (1995) Experimental results on string matching algorithms. Software: Practice & Experience 25(7):727–765. https://doi.org/10.1002/spe.4380250703

[39] Lima LG, Soares-Neto F, Lieuthier P, Castor F, Melfe G, Fernandes JP (2016) Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In: Proc. 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp 517–528, https://doi.org/10.1109/SANER.2016.85

[40] Liu J, Zhang Y, Gong B (2023) A data compression and encryption method for green edge computing. Cluster Computing 26:3341–3359. https://doi.org/10.1007/s10586-023-03968-1

[41] Mahajan D, Blakeney C, Zong Z (2019) Improving the energy efficiency of relational and NoSQL databases via query optimizations. Sustainable Computing: Informatics and Systems 22:120–133. https://doi.org/10.1016/j.suscom.2019.01.017

[42] Manber U (1997) A text compression scheme that allows fast searching directly in the compressed file. ACM Transactions on Information Systems 15(2):124–136. https://doi.org/10.1145/248625.248639

[43] Manber U, Wu S (1994) Glimpse: A tool to search through entire file systems. In: USENIX Winter, pp 1–11

[44] Mancebo J, Arriaga HO, García F, Moraga MA, de Guzmán IGR, Calero C (2018) EET: A device to support the measurement of software consumption. In: Proc. 6th International Workshop on Green and Sustainable Software (GREENS), pp 16–22, https://doi.org/10.1145/3194078.3194081

[45] Mancebo J, Calero C, Garcia F, Moraga M, Guzmán I (2021) FEETINGS: Framework for Energy Efficiency Testing to Improve eNvironmental Goal of the Software. Sustainable Computing: Informatics and Systems 30:100558. https://doi.org/10.1016/j.suscom.2021.100558

[46] Moffat A (1989) Word-based text compression. Software: Practice & Experience 19(2):185–198. https://doi.org/10.1002/spe.4380190207

[47] Moffat A, Katajainen J (1995) In-place calculation of minimum-redundancy codes. In: Proc. 4th International Workshop on Algorithms and Data Structures (WADS), pp 393–402, https://doi.org/10.1007/3-540-60220-8_79

[48] Moffat A, Stuiver L (2000) Binary interpolative coding for effective index compression. Information Retrieval 3(1):25–47. https://doi.org/10.1023/A:1013002601898

[49] Moffat A, Zobel J (1992) Coding for compression in full-text retrieval systems. In: Proc. Data Compression Conference, 1992., pp 72–81, https://doi.org/10.1109/DCC.1992.227474

[50] Silva de Moura E, Navarro G, Ziviani N, Baeza-Yates R (2000) Fast and flexible word searching on compressed text. ACM Transactions on Information Systems 18(2):113–139. https://doi.org/10.1145/348751.348754

[51] Navarro G, Raffinot M (1998) A bit-parallel approach to suffix automata: Fast extended string matching. In: Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 1448, pp 14–33, https://doi.org/10.1007/BFb0030778

[52] Navarro G, Raffinot M (1999) A general practical approach to pattern matching over ziv-lempel compressed text. In: Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 1645, pp 14–36, https://doi.org/10.1007/3-540-48452-3_2

[53] Navarro G, Raffinot M (2002) Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences. Cambridge University Press, USA, iSBN 0-521-81307-7. 280 pages.

[54] Navarro G, Raffinot M (2004) Practical and flexible pattern matching over ziv–lempel compressed text. Journal of Discrete Algorithms 2(3):347–371. https://doi.org/10.1016/j.jda.2003.12.002

[55] Navarro G, Tarhio J (2000) Boyer-moore string matching over ziv-lempel compressed text. In: Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM). Springer-Verlag, LNCS 1848, pp 166–180, https://doi.org/10.1007/3-540-45123-4_16

[56] Navarro G, Tarhio J (2005) LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. Software: Practice & Experience 35(12):1107–1130. https://doi.org/10.1002/spe.663, relevant software available at http://www.dcc.uchile.cl/~gnavarro/software/lzgrep.tar.gz.

[57] Ottaviano G, Venturini R (2014) Partitioned elias-fano indexes. In: Proc. 37th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), pp 273–282, https://doi.org/10.1145/2600428.2609615

[58] Penzenstadler B, Raturi A, Richardson D, Calero C, Femmer H, Franch X (2014) Systematic mapping Study on Software Engineering for Sustainability (SE4S). ACM International Conference Proceeding Series https://doi.org/10.1145/2601248.2601256

[59] Pereira R, Couto M, Saraiva Ja, Cunha J, Fernandes JaP (2016) The influence of the java collection framework on overall energy consumption. In: Proc. 5th International Workshop on Green and Sustainable Software (GREENS), pp 15–21, https://doi.org/10.1145/2896967.2896968

[60] Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, Saraiva J (2021) Ranking programming languages by energy efficiency. Science of Computer Programming 205:102609. https://doi.org/10.1016/j.scico.2021.102609

[61] Pinto G, Castor F (2017) Energy efficiency: A new concern for application software developers. Communications of the ACM 60(12):68–75. https://doi.org/10.1145/3154384

[62] Pinto G, Castor F, Liu YD (2014) Understanding energy behaviors of thread management constructs. In: Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications, pp 345–360, https://doi.org/10.1145/2660193.2660235

[63] Procaccianti G, Fernández H, Lago P (2016) Empirical evaluation of two best practices for energy-efficient software development. Journal of Systems and Software 117:185–198. https://doi.org/10.1016/j.jss.2016.02.035

[64] Ramírez T, Pajuelo A, Santana OJ, Valero M (2007) Energy saving through a simple load control mechanism. ACM SIGARCH Computer Architecture News 35(4):29—36. https://doi.org/10.1145/1327312.1327318

[65] Schwartz ES, Kallick B (1964) Generating a canonical prefix encoding. Communications of the ACM 7(3):166–169. https://doi.org/10.1145/363958.363991

[66] Trefethen AE, Thiyagalingam J (2013) Energy-aware software: Challenges, opportunities and strategies. Journal of Computational Science 4(6):444–449. https://doi.org/10.1016/j.jocs.2013.01.005

[67] Weaver VM, Terpstra D, McCraw H, Johnson M, Kasichayanula K, Ralph J, Nelson J, Mucci P, Mohan T, Moore S (2013) Papi 5: Measuring power, energy, and the cloud. In: Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 124–125, https://doi.org/10.1109/ISPASS.2013.6557155

[68] Williams HE, Zobel J (1999) Compressing Integers for Fast File Access. The Computer Journal 42(3):193–201. https://doi.org/10.1093/comjnl/42.3.193

[69] Witten I, Moffat A, Bell T (1999) Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers

[70] Witten IH, Bell TC, Nevill-Manning CG (1991) Models for compression in full-text retrieval systems. In: Proc. Data Compression Conference, pp 23–32, https://doi.org/10.1109/DCC.1991.213370

[71] Xu Z, Tu YC, Wang X (2015) Online Energy Estimation of Relational Operations in Database Systems. IEEE Transactions on Computers 64(11):3223–3236. https://doi.org/10.1109/TC.2015.2394309

[72] Yuki T, Rajopadhye S (2014) Folklore Confirmed: Compiling for Speed = Compiling for Energy. In: Proc. International Workshop on Languages and Compilers for Parallel Computing (LCPC), pp 169–184, https://doi.org/10.1007/978-3-319-09967-5_10

[73] Zhang J, Liu H, Ni L (2020) A Secure Energy-Saving Communication and Encrypted Storage Model Based on RC4 for EHR. IEEE Access 8:38995–39012. https://doi.org/10.1109/ACCESS.2020.2975208

[74] Ziv J, Lempel A (1977) A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3):337–343. https://doi.org/10.1109/TIT.1977.1055714

[75] Ziv J, Lempel A (1978) Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5):530–536. https://doi.org/10.1109/TIT.1978.1055934

[76] Zukowski M, Heman S, Nes N, Boncz P (2006) Super-scalar ram-cpu cache compression. In: Proc. 22nd International Conference on Data Engineering (ICDE). IEEE Computer Society, USA, p 59, https://doi.org/10.1109/ICDE.2006.150, URL https://doi.org/10.1109/ICDE.2006.150