Journal of Computational Information Systems 11: 24 (2015) 7747–7755
Available at http://www.Jofcis.com

Efficient Decoding of Posting Lists with SIMD Instructions*

Naiyong AO [1], Xiaoguang LIU [2], Gang WANG [1,*]

[1] *College of Computer and Control Engineering, Nankai University, Tianjin 300071, China*

[2] *College of Software, Nankai University, Tianjin 300071, China*

Abstract

To achieve fast query processing, search engines generally store their posting lists in main memory in a compressed format. The integer decoding of posting lists must be performed on the fly for every uncached query and consumes considerable CPU time. Therefore, efficient integer decoding algorithms are essential for search engine performance, and have been studied extensively in the literature. Recent research work has shown that byte-aligned and frame-based codecs are particularly amenable to parallel decoding with powerful SSE instructions in modern processors. In this paper, we apply these instructions to bit- and word-aligned codecs and, in particular, we exploit the wider bit width and more powerful instructions of Intel AVX2 to further improve the decoding speed. Our experiments on the TREC GOV2 collection demonstrate significant performance gains from applying AVX2 instructions to decoding posting lists.

Keywords: Inverted Index; Integer Decoding; SIMD

1 Introduction

Major web search engines process thousands of queries per second over billions of documents [1]. The rapid growth of data sizes and query loads raises enormous performance challenges to the search engines. To deal with the heavy workload, search engines use many optimization techniques such as caching and data compression. We will focus on index compression in this paper.

The most widely used data structure in text search engines is the *inverted index* [3], where, for each unique indexed term t of the collection, we store a *posting list* $\ell(t)$ containing the set of documents in which the term appears and possibly additional information. Each document is typically identified by a unique 32-bit *integer document identifier* (docID) from $\{0, 1, \ldots, N-1\}$, where N is the number of documents in the collection. In this paper, we exclusively focus on encoding and decoding the docIDs, i.e., we assume posting lists comprise entirely of docIDs.

Because posting lists can be very long and account for a large fraction of storage, some form of compression is needed to store them. The standard way of representing a posting list $\ell(t)$ is

*This work was partially supported by NSF of China (No. 61373018, 11301288), Program for New Century Excellent Talents in University (No. NCET130301), and the Fundamental Research Funds for the Central Universities (No. 65141021).

*Corresponding author.
Email address: wgzwp@nbjl.nankai.edu.cn (Gang WANG).

1553–9105 / Copyright © 2015 Binary Information Press
DOI: 10.12733/jcis15C0240
December 15, 2015

to sort it in strictly increasing order of docIDs, and then convert it to a sequence of *d-gaps* $\Delta\ell(t)$ by keeping only the differences between successive docIDs in $\ell(t)$, together with the initial value. The d-gaps are nonnegative integers that are on average much smaller than the original docIDs. Therefore, they can be compressed more effectively. We will focus on compression schemata for 32-bit nonnegative integer sequences for the remainder of this paper.

There are two important aspects to consider when evaluating the performance of any compression scheme (or codec):

- *Space-effectiveness* (i.e., compression ratio). This depends only on the encoding, and not the implementation. A codec should be space-effective to ensure that as much as possible of the inverted index can be kept in higher levels of the memory hierarchy. For example, Baidu, which is currently the dominant search engine in Chinese, compresses the most frequently accessed inverted indexes and stores them in main memory [2], as in our experiments.
- *Time-efficiency* (i.e., decoding speed). This may vary for different implementations and platforms. A codec should be time-efficient, since the integer decoding of posting lists must be performed on the fly for every uncached query, consuming considerable CPU time. However, the encoding of a codec can be relatively slow since the encoding of a posting list is only performed infrequently, at indexing time.

Various compression techniques have been proposed in the literature to achieve a better tradeoff between the space-effectiveness and time-efficiency. Recent research investigates the use of SSE instructions in modern processors to develop parallel decoding algorithms for byte-aligned and frame-based codecs. In this paper, we apply these instructions to bit- and word-aligned codecs and, in particular, we exploit the wider bit width and more powerful instructions of Intel AVX2 to further improve the decoding speed.

The remainder of this paper is organized as follows. We discuss related work in Section 2. Section 3 presents SIMD-based decoding algorithms for various codecs. Section 4 gives a summary of our experimental results. Finally, Section 5 concludes the paper and discusses future work.

2 Related Work

Below we briefly review the codecs that we investigate in our experiments.

2.1 Bit-aligned codecs

If the docIDs in $\ell(t)$ is assumed to be chosen uniformly at random from $\{0, 1, \ldots, N-1\}$, then the d-gaps in $\Delta\ell(t)$ will conform to a geometric distribution [3]. In that case, a *Golomb code* [4] is an optimal prefix code. In Golomb coding, for some parameter b, we first split any integer x into two parts: the quotient $q = \lfloor x/b \rfloor$ and the remainder $r = x - qb$. Then x is represented as $q + 1$ in unary, followed by r coded in binary, requiring either $\lfloor \log_2 b \rfloor$ or $\lceil \log_2 b \rceil$ bits. The parameter b is typically chosen to be $b = 0.69 \times avg$, where avg is the average of the integers to be coded.

Golomb decoding incurs significant branch misprediction overhead and needs to perform costly integer multiplication operations. These can be addressed by restricting the parameter b to be a power of 2. The resulting algorithm is known as *Rice coding* [5].

2.2 Byte-aligned codecs

Byte-aligned codecs can be traced back to *varint-SU* [6]. In varint-SU, an integer is coded as a sequence of bytes, where each byte consists of one descriptor bit, indicating whether the current byte is the last one, followed by 7 data bits. When decoding, we read the bytes sequentially, and output a new integer whenever the descriptor bit is 1, leading to costly branch mispredictions when the integers are large. *Varint-GB* [1] encodes a group of four integers as a descriptor byte followed by the data bytes. The descriptor byte contains four 2-bit binary descriptors, representing the lengths of the corresponding integers in the group. Compared with varint-SU, varint-GB achieves slightly worse compression but more efficient decoding with the use of a lookup table.

Both of the above codecs fall into the taxonomy defined by Stepanov et al. [7]. Using the taxonomy, they described a new codec, *varint-GU*, which differs from varint-GB in that it operates on a group of 9 bytes, with 8 data bytes encoding 2 to 8 integers, and 1 descriptor byte representing the lengths of the integers in unary. Two variations, *varint-G8IU* and *varint-G8CU*, were also classified depending on whether the codeword of an integer can span across consecutive groups.

2.3 Word-aligned codecs

Word-aligned codecs encode as many integers as possible in a single machine word. Among them are *Simple-9* [8] and *Simple-16* [9], which operate on 32-bit words.

In Simple-9, a 32-bit word is divided into two parts, a 4-bit selector and 28 data bits. There are nine possible ways in which the 28 data bits can be partitioned into equal length binary codes, together with possibly unused bits. For example, if the next 3 integers are less than 512, then we can store them as three 9-bit binary codes, leaving one data bit unused. The selector value describes which of the nine partitions is being used in the word. Simple-16 extends Simple-9 by using the spare selector values to describe asymmetric combinations.

2.4 Frame-based codecs

Frame-based codecs have been shown to offer the highest throughput in a search engine setting. Among them are *NewPFor* and *OptPFor* [10].

NewPFor divides the sequence of integers into segments of length k, for some k divisible by 32. For each segment A, the smallest $b = b(A)$ is determined such that most integers (say 90%) in A are less than 2^b, while the remainder are *exceptions*. The non-exceptions in A, along with the least significant b bits of each exception, are stored in k b-bit slots. The most significant $32-b$ bits of the exceptions (called the *overflow*), along with the offsets between consecutive exceptions, are stored in two separate arrays, which are further compressed using Simple-16. OptPFor, a variant of NewPFor, works similarly but selects the value of b that maximizes the compression ratio.

3 SIMD Decoding

In this section, we present our SIMD-based decoding algorithms.

3.1 Decoding unary codes

A straightforward way of decoding unary codes is to read the bits sequentially, and output an integer whenever a terminating one bit is encountered. This incurs significant branch mispredictions and bit manipulations, which can be largely eliminated by using a lookup table of 2^{16} entries. The decoding is still inefficient because it incurs too many memory reads (twice the number of 32-bit words consumed). A lookup table of 2^{32} entries, however, cannot fit into the CPU cache.

At the heart of our approach there is an efficient decoding algorithm, *decodeWord*, that operates on a single 32-bit word. The details are shown in Algorithm 1. The algorithm takes three inputs:

- *src* – a pointer to the input 32-bit word stream of encoded values.
- *dst* – a pointer to the output stream of integers.
- *carry* – the number of leading 0 bits in the prior 32-bit word of *src*.

The algorithm reads a 32-bit *word* from the input stream, outputs decoded integers to the output stream, and returns as its result the new position of *src*, *dst*, and the updated *carry*.

The algorithm depends on the following abstract functions:

- popcount(*word*) gives the number of integers whose codeword is complete in *word*. This is equal to the number of 1 bits in *word*.
- ctz(*word*) gives the value of the first encoded integer in *word*. This is equal to the number of trailing zeros in *word*.
- clz(*word*) gives the number of bits that should be carried over to the next 32-bit word. This is equal to the number of leading 0 bits in *word*.

These three functions are implemented as the GCC built-in functions, `__builtin_popcount`, `__builtin_ctz`, and `__builtin_clz`, respectively. Those functions have constant complexity, so the total running time of decodeWord is linear in the number of 1s in the word.

Algorithm 1 decodeWord

```
Input: $src$, $dst$, $carry$
Output: $src$, $dst$, $carry$
 1: begin
 2:   $word \leftarrow src[0]$
 3:   $k \leftarrow \text{popcount}(word)$
 4:   switch ($k$)
 5:   case 0:
 6:     $carry \leftarrow carry + 32$
 7:     break
 8:   case 32:
 9:     memset($dst$, 0, 32)
10:     $dst[0] \leftarrow carry$
11:     $carry \leftarrow 0$
12:     break
13:   default:
14:     for $0 \le i < k$ do
15:       $dst[i] \leftarrow \text{ctz}(word)$
16:       $word \leftarrow word \gg (dst[i] + 1)$
17:     end for
18:     $dst[0] \leftarrow dst[0] + carry$
19:     $carry \leftarrow \text{clz}(src[0])$
20:     break
21:   end switch
22:   $src \leftarrow src + 1$
23:   $dst \leftarrow dst + k$
24:   return $src, dst, carry$
25: end
```

3.2 Vectorized bit unpacking

Given a block of s b-bit integers $\{v_0, v_1, \dots v_{s-1}\}$, there are two different ways to store them:

- *Horizontal layout* [12] stores the integers according to their order. Fig. 1 illustrates an example where b is equal to 11.

- *k-way vertical layout* [11] assigns k consecutive integers to k different groups, where each group corresponds to a 32-bit word. In Fig. 2 we present an illustrative example for a 4-way vertical layout where b is equal to 11.

Both of them are particularly amenable to SIMD-based decoding, especially the latter.

3.2.1 Vectorized horizontal bit unpacking

Fig. 1 illustrates the core steps of the SSE-based decoding algorithm for the horizontal layout:

- *Shuffle* – The four compressed values $\{v_0, v_1, v_2, v_3\}$ are copied into four separate 32-bit Double-Words (DW) of a SIMD register using a shuffle instruction for bytes.
- *Align* – The four values are shifted to the right in order to align them. This is realized by multiplications by powers of two followed by shifting all values by the same shift amount.
- *Clean* – A mask is applied to clean the remaining $32 - b$ bits in the four DWs.

The eight values $\{v_{4i}, v_{4i+1}, \ldots, v_{4i+7}\}$ are called a block. Note that the first codeword of a block is always aligned to a byte boundary, so the shuffle sequences, the multiplication sequences and the shift amounts are constant across all blocks.

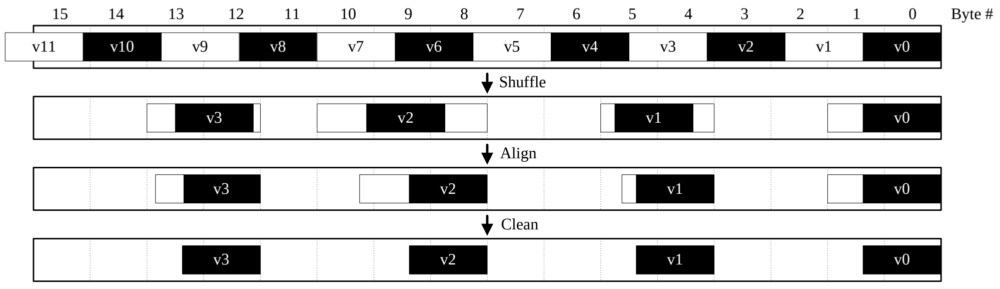


Fig. 1: SSE-based horizontal bit unpacking

The details of constructing the shuffle sequences are shown in Algorithm 2. The algorithm starts by initializing the shuffle sequences shf to all -1, and then computes for each value in the block its corresponding shuffle sequence. The codeword of the i-th value spans from bit position ib to bit position $(i+1)b-1$, so the i-th shuffle sequence has the form

$$\overbrace{(\lfloor ib/8 \rfloor, \lfloor ib/8 \rfloor + 1, \ldots, \lfloor ((i+1)b-1)/8 \rfloor, -1, -1, \ldots, -1)}^{\text{length } 4}.$$

In the example of Fig. 1, the shuffle sequence would be

$$\{\overbrace{(0,1,-1,-1)}^{v_0}, \overbrace{(1,2,-1,-1)}^{v_1}, \overbrace{(2,3,4,-1)}^{v_2}, \overbrace{(4,5,-1,-1)}^{v_3}\}.$$

If a_i denotes the number of leading invalid bits in the first four DWs, then to align the DWs, we multiply the i-th DW by $2^{(\max_i a_i)-a_i}$ and then right-shift by $\max_i a_i$. To facilitate this computation, the values of $2^{(\max_i a_i)-a_i}$ and $\max_i a_i$ are precomputed and stored. The second four DWs are similarly aligned. Algorithm 3 gives the details of computing the multiplication sequences and the shift amounts. In the example of Fig. 1, the multiplication sequences would be $(2^6, 2^3, 2^0, 2^5)$, while the shift amount is 6.

The SSE-based horizontal bit unpacking algorithm almost directly translates to the Intel AVX2 instructions. In particular, the align step can be realized more efficiently with the AVX2 `vpsrld` instruction, which supports simultaneously shifting several values by different offsets.

Algorithm 2 constructShuffleSequence

```
Input: $b$
Output: $shf$
begin
    for $0 \le k < 32$ do
        $shf[k] \leftarrow -1$
    end for
    for $0 \le i < 8$ do
        $k \leftarrow 4i$
        $beg \leftarrow \lfloor ib/8 \rfloor$
        $end \leftarrow \lfloor ((i+1)b-1)/8 \rfloor$
        for $beg \le j \le end$ do
            $shf[k] \leftarrow j$
            $k \leftarrow k+1$
        end for
    end for
    return $shf$
end
```

Algorithm 3 constructAlignSequence

```
Input: $b$
Output: $mul, sht$
begin
    for $0 \le k < 8$ do
        $mul[k] \leftarrow (kb)\%8$
    end for
    for $0 \le i < 2$ do
        $j \leftarrow 4i$
        $m \leftarrow \max_{j \le k < j+4}(mul[k])$
        for $j \le k < j+4$ do
            $mul[k] \leftarrow 1 \ll (m - mul[k])$
        end for
        $sht[i] \leftarrow m$
    end for
    return $mul, sht$
end
```

3.2.2 Vectorized vertical bit unpacking

The SSE-based decoding algorithm for the vertical layout works similarly, but does not need to perform the shuffle operation. Fig. 2 illustrates the core steps of decoding the four compressed values $\{v_4, v_5, v_6, v_7\}$. All four values are first shifted to right by 11 bits and then masked. For the four values $\{v_8, v_9, v_{10}, v_{11}\}$, however, a SSE OR instruction must be performed to concatenate the codewords distributed across consecutive groups. This algorithm directly translates to the Intel AVX2 instructions as long as a 8-way vertical layout is used.

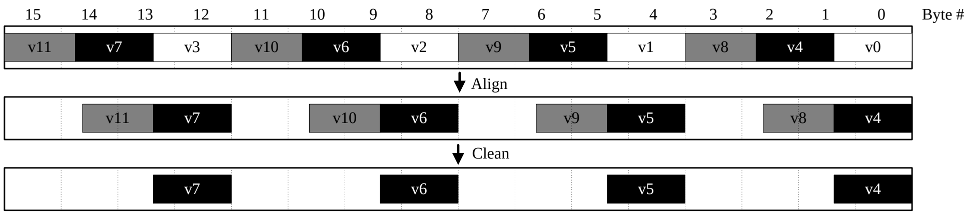


Fig. 2: SSE-based vertical bit unpacking

4 Experiments

We ran all our experiments using a single-threaded process on an Intel Core i7-4700K processor (3.5 GHz, 4 cores, 6 MB Intel Smart Cache shared across all cores), with 32 GB DDR3 1600 MHz RAM. The operating system is a 64-bit Linux with kernel version 2.6.32-431. Measurements are done with all of the input and output data in main memory. We implemented our algorithms in C++ using GNU GCC 4.9.2 with the optimization flag -O3.

We use the TREC *GOV2* data set, which contains 25 million documents. The docIDs are the original docIDs, and we do not have access to how they are assigned. We use the method proposed in [13] to reorder the docIDs alphabetically according to URLs, and form a dataset *GOV2URL*. We form another dataset *GOV2R* by randomly assigning docIDs [2].

4.1 Compression ratio

For each data set, Table 1 shows the compression ratio measured in bits per integer; the best results are shown in boldface. Compression ratios depend only on the encoding, and not the data layout and platform. We introduce a new codec, *OptRice*, which differs from Rice in that it selects the parameter b that maximizes the compression ratio. We also introduce two variations of varint-GB, *varint-G4B*, which uses one descriptor byte and encodes 4 values in a group, and *varint-G8B*, which uses two descriptor bytes and encodes 8 values in a group.

Unsurprisingly, the best compression ratio on GOV2R is obtained with OptRice; it also happens to be true on GOV2. The difference between Rice and OptRice is minor, which indicates that the choice of b-value in Rice leads to nearly optimal results. Byte-aligned codecs require codewords to end on byte boundaries, to which we attribute their poor compression performance over all datasets; the loss percentage differs by more than 100 percentage points on GOV2URL. NewPFor is competitive with all the other codecs over all datasets; the loss percentage differs by at most 15.7 percentage points. All the codecs perform better on GOV2URL than the other two datasets, indicating that all of them benefit from clustered docIDs.

Table 1: Compression ratios in bits per integer (bpi); smaller is better

categories	codecs	GOV2		GOV2URL		GOV2R	
		bpi	loss %	bpi	loss %	bpi	loss %
bit-aligned	Rice	7.64	1.1	5.96	24.7	8.13	1.1
	OptRice	**7.56**	0	5.90	23.4	**8.04**	0
byte-aligned	varint-G4B	11.49	52.1	10.67	123.1	11.90	47.9
	varint-G8B	11.52	52.4	10.69	123.5	11.92	48.2
	varint-G8IU	10.88	44.0	9.86	106.2	11.43	42.1
	varint-G8CU	10.69	41.5	9.77	104.2	11.15	38.6
word-aligned	Simple-9	9.32	23.4	5.22	9.2	10.24	27.3
	Simple-16	8.85	17.2	4.99	4.3	9.79	21.7
frame-based	NewPFor	8.07	6.9	5.53	15.7	8.58	6.7
	OptPFor	7.82	3.5	**4.78**	0	8.38	4.2

4.2 Decoding speed

For each data set, Table 2 shows the decoding speed measured in millions integers per second; the best results are shown in boldface. We use the suffix -Hor and -Ver to represent the horizontal layout and vertical layout, respectively.

Unsurprisingly, bit-aligned codecs generally achieve the slowest decoding speeds; the only exceptions are that Simple-16-AVX2 is slightly slower than Rice-Ver-AVX2 on GOV2, and slightly slower than Rice-Ver-AVX2 and OptRice-Ver-AVX2 on GOV2URL. Of the bit-aligned codecs, the codecs that use AVX2 and a vertical layout have better decoding speeds; on GOV2 and GOV2R, Rice-Ver-AVX2 is $\geq 6.4\%$ faster than Rice-Ver-SSE, and $\geq 26.9\%$ faster than Rice-Hor-AVX2.

Varint-G8IU-AVX2 offers the best decoding speeds over all datasets. The scalar version of varint-GU is much slower than varint-GB; varint-G8B-Scalar is roughly twice as fast as varint-G8CU-Scalar. The vectorized version of varint-GU, however, is much faster than varint-GB; varint-G8IU-AVX2 is around 3.6 times faster than varint-G4B-AVX2 on GOV2URL.

We can also find that NewPFor is much faster than OptPFor; NewPFor-Ver-AVX2 is around 2.5 times faster than OptPFor-Ver-AVX2 on GOV2URL. Frame-based codecs based on AVX2 and a vertical layout are preferable; NewPFor-Ver-AVX2 is 11% faster than NewPFor-Ver-SSE on GOV2 and 29% faster than NewPFor-Hor-AVX2 on GOV2R.

In conclusion, all our bit-aligned and frame-based codecs benefit from the Intel AVX2 instructions and a vertical layout.

Table 2: Decoding speeds in millions of integers per second (mis); larger is better

codecs	GOV2			GOV2URL			GOV2R		
	Scalar	SSE	AVX2	Scalar	SSE	AVX2	Scalar	SSE	AVX2
Rice-Hor	715	739	751	787	880	897	719	744	758
Rice-Ver	709	897	954	781	1005	1056	715	902	962
OptRice-Hor	701	723	730	766	840	843	706	731	737
OptRice-Ver	687	878	924	735	955	994	696	886	933
varint-G4B	1320	1483	1303	1322	1482	1310	1321	1481	1317
varint-G8B	1754	2035	2028	1756	2034	2023	1759	2036	2029
varint-G8IU	1204	4381	**4403**	1216	4692	**4740**	1183	4191	**4374**
varint-G8CU	880	4170	4130	900	4280	4483	867	4034	4134
Simple-9	989	1063	1058	1172	1347	1344	970	1030	1022
Simple-16	889	930	940	1170	1317	1328	882	912	920
NewPFor-Hor	1930	2794	3146	1896	2781	3030	2040	2894	3273
NewPFor-Ver	1905	3625	4022	1907	3400	3716	1999	3834	4222
OptPFor-Hor	1204	1555	1709	1022	1281	1331	1272	1638	1819
OptPFor-Ver	1193	1877	2046	1051	1427	1498	1251	2004	2223

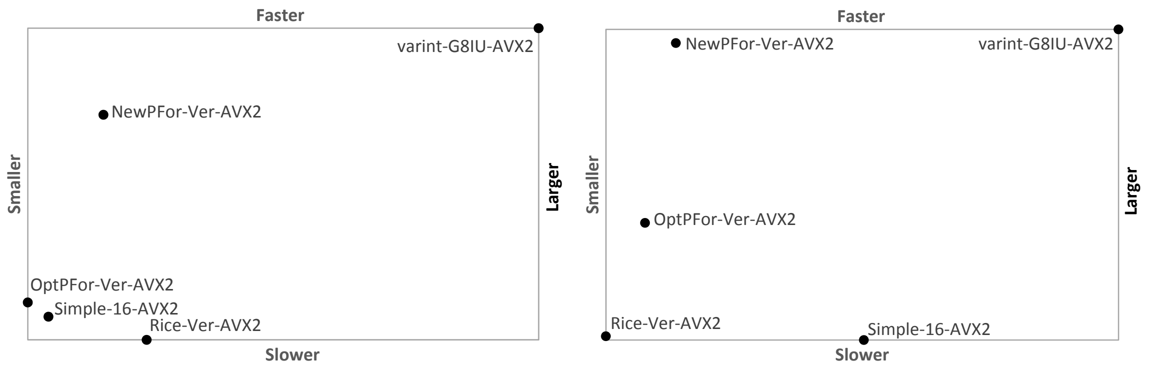
4.3 Tradeoff

Ideally, a codec would achieve both a high compression ratio and fast decoding speed. However, in practice, there is a tradeoff between these two aspects. In Fig. 3 we plot the normalized compression ratio versus normalized decoding speed for representative codecs from the four categories. Since our codecs perform similarly on GOV2 and GOV2R, we only include a plot for GOV2R.

We can see that varint-G8IU is the fastest but the least effective. For Rice-Ver-AVX2, Simple-16-AVX2, and OptPFor-Ver-AVX2, while we see decent space-effectiveness, they are the slowest. NewPFor-Ver-AVX2 simultaneously achieves a high compression ratio and fast decoding speed.

5 Conclusions and Future Work

We investigated the use of SIMD instructions in modern processors to develop parallel decoding algorithms for search engine posting lists. In particular, we exploit the wider bit width and more



(a) GOV2URL (b) GOV2R

Fig. 3: Tradeoff between space-effectiveness and time-efficiency

powerful instructions of Intel AVX2. Our experiments on the TREC GOV2 collection demonstrate significant performance gains from applying AVX2 instructions to decoding posting lists.

References

[1] J. Dean. Challenges in building large-scale information retrieval systems. Keynote. Proc. 2nd ACM International Conference on Web Search and Data Mining, 2009.

[2] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, and X. Liu. Efficient parallel lists intersection and index compression algorithms using graphics processing units. Proc. VLDB Endow., 4(8), pages 470-481, 2011.

[3] I. H. Witten, A. Moffat, and T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, second edition, 1999.

[4] S. Golomb. Run-length encodings. IEEE Transactions on Information Theory, 12(3): 399-401, 1966.

[5] R. F. Rice, and J. R. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. IEEE Transactions on Commununication Technology, 19(6): 889-897, 1971.

[6] L. H. Thiel, H. S. Heaps. Program design for retrospective searches on large data bases. Information Storage and Retrieval, 8(1): 1-20, 1972.

[7] A. A. Stepanov, A. R. Gangolli, and D. E. Rose. SIMD-based decoding of posting lists. Proc. 20th ACM International Conference on Information and Knowledge Management, pages 317-326, 2011.

[8] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. Information Retrieval, 8(1): 151-166, 2005.

[9] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. Proc. 17th International Conference on World Wide Web, pages 387-396, 2008.

[10] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. Proc. 18th International Conference on World Wide Web, pages 401-410, 2009.

[11] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. Software: Practice and Experience, 45(1): 1-29, 2015.

[12] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, and A. Zeier. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. Proc. VLDB Endow., 2(1): 385-394, 2009.

[13] F. Silvestri. Sorting out the document identifier assignment problem. Proc. 29th European Conference on Information Retrieval, pages 101-112, 2007.