SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:131–147
Published online 12 January 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.948

Index compression using 64-bit words

Vo Ngoc Anh and Alistair Moffat[*, †]

Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia

SUMMARY

Modern computers typically make use of 64-bit words as the fundamental unit of data access. However the decade-long migration from 32-bit architectures has not been reflected in compression technology, because of a widespread assumption that effective compression techniques operate in terms of bits or bytes, rather than words. Here we demonstrate that the use of 64-bit access units, especially in connection with word-bounded codes, does indeed provide the opportunity for improving the compression performance. In particular, we extend several 32-bit word-bounded coding schemes to 64-bit operation and explore their uses in information retrieval applications. Our results show that the Simple-8b approach, a 64-bit word-bounded code, is an excellent self-skipping code, and has a clear advantage over its competitors in supporting fast query evaluation when the data being compressed represents the inverted index for a large text collection. The advantages of the new code also accrue on 32-bit architectures, and for all of Boolean, ranked, and phrase queries; which means that it can be used in any situation. Copyright © 2010 John Wiley & Sons, Ltd.

Received 30 July 2008; Revised 8 September 2009; Accepted 15 September 2009

KEY WORDS: performance; measurement; index compression; information retrieval

1. INTRODUCTION

Information retrieval systems such as web search engines are, in no small part, able to handle large volumes of data because of the use of suitable compression techniques; and are, again in no small part, able to resolve queries quickly because the compression methods used allow fast decoding. Zobel and Moffat [1], Witten *et al.* [2], and Ziviani *et al.* [3] all describe this reliance. Another factor—naturally—in determining query performance is the underlying machine architecture, with hardware advances providing the basis for much of the performance improvement that has been achieved over recent years. In this environment, it is perhaps surprising to see that technological progress has had little (if any) direct influence on index compression techniques. The only architectural factor that has been routinely noted when discussing compression is that storage requirements have become, to a certain extent, relatively less important than the decompression speed. Hence, compression methods, such as variable-length Byte codes (see, for example, Scholer *et al.* [4] and Trotman [5]), which are less effective but more efficient than the traditional codes, such as the Golomb code (see the descriptions by Golomb [6], Zobel and Moffat [7], and Manning

*Correspondence to: Alistair Moffat, Department of Computer Science and Software Engineering, The University of Melbourne, Victoria 3010, Australia.
†E-mail: alistair@csse.unimelb.edu.au

Contract/grant sponsor: Australian Research Council
Contract/grant sponsor: ARC Center for Perceptive and Intelligent Machines in Complex Environments
Contract/grant sponsor: Google Research Awards grant

et al. [8], and see the work of Büttcher and Clarke [9] for a variant that applies to words-only inverted files) are argued to be superior for practical purposes.

This paper explores the use of 64-bit machine words in index compression applications. The absence of prior work is readily understandable, because compression normally means dealing with bit or byte operations, and hence both compression effectiveness and compression efficiency would seem to be unaffected by word length. Our work demonstrates that this intuition regarding the performance of compression methods is not necessarily true. In particular, we show that the Simple, Carry, and Slide coding schemes described by Anh and Moffat [10, 11], referred to here as *word-bounded codes*, present significant opportunities for enhancement under 64-bit computing regimes, in terms of both compression effectiveness and decompression efficiency.

To verify our claims, we have measured the retrieval speed for a realistic query sequence against a large text collection using a variety of compression methods, and for a range of query modalities. The 64-bit methods introduced in this paper provide excellent performance in all of the text retrieval situations tested, and represent an attractive compromise between speed and space. We also show that the new methods are appropriate even while using 32-bit machine architectures.

Section 2 summarizes the range of text query modalities, and describes the underlying index access operations that are necessary to support them in the current implementation approaches. Section 3 then describes a range of integer codes of the type that are used to represent inverted indexes, and evaluates the extent to which they intrinsically support the required access modes. Section 4 focuses on the word-bounded codes that are the primary interest in this paper, and extends them to 64-bit architectures. In Section 5, the important 'forwards seek' operation is discussed, and the advantages of word-bounded codes are explored. A range of codes are then tested on synthetic and real data in Section 6. Using a large web collection, our proposed methods are compared with other coding techniques in terms of both index storage cost and query execution speed for all Boolean, ranked, and phrase queries. Section 7 discusses an alternative fast in-memory decode technique that can be applied to a range of codes.

2. TEXT QUERY PROCESSING

An information retrieval system maintains a collection of text documents, and, on receiving a query from a user, retrieves the documents that best match the query. This computation is carried out using an *inverted index*, which records, for each distinct word or term, the list of documents that contain the term, and depending on the query modalities that are being supported, may also incorporate the frequencies and impacts of each term in each document, plus a list of the positions in each document at which that word appears. For effective compression, the lists of document and position numbers are usually sorted and transformed to the corresponding sequence of differences (or *gaps*) between adjacent values. For more details of these arrangements, see Witten *et al.* [2] and Zobel and Moffat [7].

Regardless of the structure of the information stored in the index and irrespective of the exact coding method used to store it, there are certain basic processing modes that are needed to support the usual range of text retrieval query modalities. These are

- *Successor*, to decode and return the next pointer. This operation steps the file cursor (the current access point) forwards a single pointer (or, depending on the exact context, a single integer).
- *Forwards Search*, to locate the next document pointer whose value is greater than or equal to d, for some supplied value d. This operation potentially shifts the file cursor to a large distance, in terms of both pointers and stored integers.
- *Forwards Seek*, to shift the file cursor forwards by x integers, for some supplied value x. In comparison with *Forwards Search*, this operation steps a defined number of integers, rather than until a particular value is located.

The use of these three access modes is illustrated in Figure 1, in which it is supposed that a two-word phrase query has been posed, and a document can only be an answer if it contains

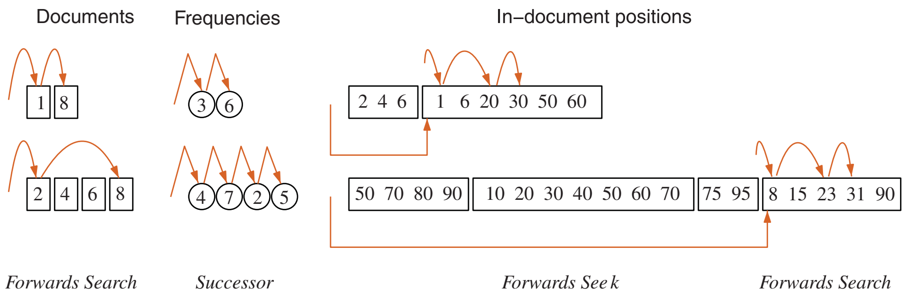

Figure 1. The different operations required when processing a two-word phrase query. Each of the two inverted lists is shown as three sections (documents, frequencies, and positions), with all values absolute (that is, without taking gaps). The processing begins with *Forwards Seek* operations in the document lists. Document 8 is discovered to appear in both lists. The *Successor* operations on frequencies show that there is no need to decode the first 3 and 13 items, respectively, in the two lists of word positions. *Forwards Seek* operations are then employed in the positions first to locate the beginning of the position sublist for document 8 in each list. Finally, *Forwards Search* operations are employed in the two position lists until it is determined that the two query words appear in positions 30 and 31, respectively, and hence constitute an answer phrase in document 8.

those two words in consecutive positions. For phrase queries, all three access modes are used, as first candidate documents containing all of the query terms (somewhere) are identified; then the corresponding position lists are located; and finally the word positions are checked.

Two other basic query modalities only require subsets of the access modes. Conjunctive Boolean queries, for which a document is an answer if it contains all of the query terms even if not as a phrase, can be resolved using just *Forwards Search*. Processing of Ranked queries in a term-at-a-time manner with dynamic pruning can be carried out using *Successor* and *Forwards Search* operations, see Zobel and Moffat [7] for a description of the techniques used to calculate heuristic similarity scores, and the way they are implemented. Alternatively, if document-at-a-time processing is being used with a document-sorted index and dynamic query pruning [12], only sequential *Successor* access is required.

The *Forwards Search* and *Forwards Seek* operations shift the file cursor, and can, of course, be implemented via a sequence of *Successor* operations. But, as can be seen in Figure 1, it is more efficient if these operations are implemented in a way that allows unneeded data elements to be bypassed rather than decoded or otherwise accessed. The fundamental difference between the two operations, in terms of potential efficiency when compression is taken into account, is that a *Forwards Seek* operation is normally used to bypass an entire segment, to reach a known re-start point for the document or word-position gaps. On the other hand, the *Forwards Search* operation typically operates within a gapped segment, and it is much harder to completely bypass values, because of the dependency of each on its predecessor.

In the past, bypass was implemented by providing true-random or pseudo-random access into the underlying inverted list. For example, if true-random access is possible, binary search or finger-based variants of it are possible [13]. Pseudo-random access for compressed lists is provided by the skipping mechanism of Moffat and Zobel [14], explored further by Strohman and Croft [15]; by Büttcher and Clarke [16]; and by Culpepper and Moffat [17]. By providing coded 'forward pointers' (and 'forward values' if *Forwards Search* is to be supported) over uniform-length blocks, the skipping mechanism makes it possible to step along a list of compressed values, and stop when the required number of pointers have been bypassed, or the block containing the desired value has been located. The drawback of these approaches is that additional information is added to the index, making it larger; and dealing with the extra values might slow down query processing rather than speed it up. Maintenance of internal structures in the inverted lists may also make it harder to undertake index modifications, such as those that arise when documents are added to the collection. Moreover, the skip length must be defined during index construction, and might not be a good match to the conditions that arise during query processing.

Another key part of this investigation was the extent to which the compression codes can provide bypassing in their own account. However, because of the gap nature of the compressed lists, the self-bypassing investigation is limited to the case of *Forwards Seek* alone. That is, we sought to determine the extent to which each of the index codes could accelerate the processing of phrase (and phrase-like) queries.

Finally in this section, note that Figure 1 shows a non-interleaved index, in which each inverted list consists of three distinct sections. In a fully interleaved index arrangement, in which the pointers sub-lists follow immediately after each document number (or gap), *Forwards Seek* is also beneficial in the Boolean and ranked query modes.

3. ARCHITECTURE-INDEPENDENT CODES

For compression purposes, and also to facilitate fast query processing, each of the term inverted lists that make up the index is best considered as being a separate data stream, and each component of a list (document, frequency, impact, position) as a non-interleaved sub-stream. Hence, the task of index compression is best viewed as being to effectively and efficiently code sequences of integers of the form $x_1, x_2, x_3, \ldots, x_n$ where $x_i \geq 1$ for all $1 \leq i \leq n$, and $n \geq 1$ is the length of the sequence.

The sequences involved in inverted index compression have quite distinctive features. They are over large alphabets; many of the possible source symbols are of low (including zero) frequency; the probability distribution generated by the taking of gaps tends to be naturally monotonicaly decreasing; and the frequencies' component tends to have a low information content. Because of these features, static integer coding methods, such as Golomb codes or Elias' Gamma and Delta codes better suit these sequences than do dictionary-based and adaptive coding methods. Operational requirements—the quest for query processing speed—reinforce this preference. Witten *et al.* [2] give comprehensive details of the structure of these sequences, and of the codes that can be used to represent them.

3.1. Code description

Most integer codes can be expressed as a composition of two elementary codes—*unary* and *binary*—and, for convenience, each primitive variable-bit unary or binary codeword component is referred to as a *snip*. For example, a Golomb codeword consists of a unary snip followed by a binary snip, and an Elias Delta codeword contains one unary and then two binary snips. For details of these standard codes, see Witten *et al.* [2].

The average number of snips per codeword can be reduced by grouping a number of consecutive codewords and forcing them to share the first snip. One simple grouping scheme, referred to here as the PackedBinary code, uses groups with fixed number of values. For each group, the least value of b that can be used to code the largest element of the group is determined and coded as a five-bit binary snip, and each of the group values as a single b-bit snip. A large group size not only reduces the average number of snips, but also leads to poorer compression because of the potential for discrepancies between the values within each group. In the experiments presented later in this paper, we use a group size of 16 with the PackedBinary method—the best space-saving value found in a series of preliminary experiments.

In the recent work, Zukowski *et al.* [18] describe an enhanced PackedBinary method which they call the PForDelta code. This method is similar to the PackedBinary method, but with a critical difference—rather than setting the value b so that all of the group values can be coded within b bits, it is set so that a large majority can be coded within b bits. The small number of exceptional values are individually coded after the group of b-bit codes, with the location of each exception needing to be identified within the block, and then those values coded, in each case making use of the b bits already coded in the main sequence to defray the cost.

The experimentation reported in this paper employs the PForDelta variant described by Zhang *et al.* [19], which was designed specifically for inverted index compression and was demonstrated to have good performance. In this variant the group size is set to 128 data elements. For each group, the large majority that can be coded within *b* bits is set so that it covers at least 90% of the group. Each of the exceptions are coded in either 8, 16, or 32 bits. For each group, four parameter values need to be recorded: *b*, the bit size for the exceptions, the number of exceptions, and the index of the first exception value. The decoding starts by extracting the four parameters; then unpacking the block of 128 *b*-bit elements, and finally decoding the exceptions.

One problem is dealing with short groups that have fewer than 128 elements each. Zhang *et al.* [19] sidestep this problem by not employing PForDelta in this case, but instead switching to a different code (Byte, which will be introduced shortly). Here, we choose a different solution to maintain the applicability of the code to short groups. In these groups, we force the parameter *b* to be large enough so that no exceptions are needed. And since there is no exception, there is no need to decode or encode the full 128 *b*-bit block. If the decoding procedure does decode a whole group of 128 elements (see Section 7), we just allow that to happen—it does not affect the correctness of any of the retrieval operations.

In the decoding algorithm of Zhang *et al.* [19], it is critical that the code of the main 128-element array begins from a machine word boundary so that the fast decoding mechanism can operate. In the experiments reported here we relax that requirement, allowing the code of each group and any part of it to begin at any bit position. This ensures that the code is set in the same context as the others that are explored here, and allows us to better isolate the effect of different computer architectures. Compared with PackedBinary, we expect PForDelta decoding to be slightly slower, because of the need to handle the exceptions, but for PForDelta to achieve slightly better compression.

Byte-aligned codes, denoted Byte, are another option for compressing inverted files [4, 5, 20, 21]. Byte codes tend to be less effective than bitwise codes, but can be decoded more quickly. In the simple Byte code, each codeword is a sequence of one-byte binary snips, with the first bit of each snip indicating whether it is the final snip of the codeword.

3.2. Decoding techniques

Traditionally, the unary and binary snips are decoded in a *bit-by-bit* manner using a one-bit buffer—see, for example, the description given by Williams and Zobel [22]. The exception is Byte, where decoding is done using a one-byte buffer, in a *byte-by-byte* manner. It is not difficult to see the advantages of Byte over bitwise decoding in terms of speed, and this differential has always been used as a key argument in favor of the Byte method. Note that with the bit-by-bit and byte-by-byte processing, variations in machine word length are unlikely to have any great effect on the decoding speed.

On the other hand, it is also possible to perform *snip-by-snip* decoding of composite bitwise codes, with one snip (instead of one bit) being decoded at a time. In this processing mode, the decoding time is proportional to the number of snips rather than the bit-length of the compressed stream as in the case of bit-by-bit decoding, and it is markedly faster than the latter [11].

To support snip-by-snip decoding the snips, as well as the bits inside each unary snip, should be ordered from the least significant to the most significant bits of the machine word in which they are stored. An example is provided shortly. A one-word buffer is then used to hold the current word of the compressed stream, and decoding operations are performed over this buffer. In the case of a binary snip being next required, a mask operation is used to extract the specified number of low-order bits as an integer. In the case of unary snip being required, a 256-element table is used to extract the value of the unary snip from the last byte of the buffer. If the snip lies fully in that byte, the value decoded is the length of the snip. If a unary snip spans more than 1 byte, the buffer is shifted 1 byte to the right, and the same process is repeated, with the decoded value accumulated. After decoding either a binary or unary snip, the consumed bits are removed via a shift operation that brings the next snip into the low-order end of the buffer. If snips are permitted to span word boundaries, the operation becomes a little bit more complicated at the end of each word, and a small number of additional operations are required to stitch the two parts of the split snip back together.

3.3. *Effect of architecture on decoding*

It might appear that the relative decoding speed for the PackedBinary, PForDelta, and Byte methods described earlier in this section should be fairly independent of the word length in the machine architecture. In fact, the fundamental snip-by-snip operation is sensitive to the length of the word unit, and thus sensitive to architecture. If a (say) binary snip spans a word boundary, the cost of decoding it is almost two-fold that of a within-a-word snip. On the other hand, although both 32-bit and 64-bit machine architectures allow operations over both *short* (32-bit) and *long* (64-bit) word units, the operations can have quite different performances. In general, we expect that 32-bit machines are fast with short word operations, and relatively slow with long word operations. Machines with 64-bit words are likely to offer the same speed on both 32-bit and 64-bit quantities.

Hence, when exploring the speed of snip-by-snip decoding, word size must be identified. In the discussion below we specify PForDelta as PForDelta-4b and PForDelta-8b to indicate that short and long words, respectively, are employed as the decoding buffer. A similar notation is applied to PackedBinary.

4. ARCHITECTURE-DEPENDENT COMPRESSION

The codes introduced in the previous section are architecture-independent in the sense that their space effectiveness is unaffected by any change of the underlying machine word length. This section briefly introduces a number of alternatives, generically referred to as *word-bounded codes* in this paper. These codes were developed under the assumption that machine words were fixed at 32 bits. That assumption is now rarely correct, and the adaptations necessary to create 64-bit versions of them are also described.

In the word-bounded codes, a certain number of data elements are bulk-coded together as a *group* [10, 11]. Each group of λ elements consists of a binary snip called a *selector*, followed by λ equal-length binary *data* snips, one for each of the values in the group. The selector specifies the common length θ of the data snips, and hence also decides the number of data elements of the group (λ), based on the requirement that all $\lambda+1$ snips fit within and maximally occupy (approximately, with some slight variations that characterize the different methods) one machine word. The length and the interpretation of the selector depend on the particular coding method.

4.1. *The* Simple *family*

In the first method using this approach, Simple-9, a 4-bit selector is used in each 32-bit machine word, leaving 28 bits available for data. Within those data bits, one 28-bit integer, or two 14-bit integers, or three 9-bit integers, and so on can be accommodated. If indeed three 9-bit numbers are stored in the word, but one of them is only a 6-bit number, then three bits have been wasted as a result of the aggregating of data elements into groups. Another bit (in this example) is also wasted because $9\times 3=27$ is one bit less than the 28 bits nominally available. Figure 2 illustrates this simple example in more detail, and shows the layout of the selector and the data bits within the machine word storing the group. This simplest word-bounded method was denoted by

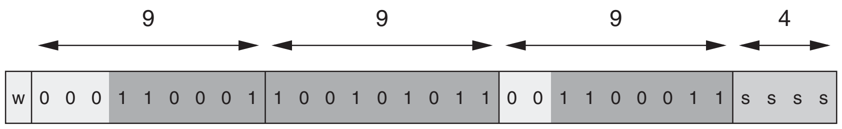
9
9
9
4
w 0 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 0 0 1 1 s s s s

Figure 2. Using the Simple-4b method to store the three integers 100, 300, 50 (represented as 99, 299, 49 since the values are presumed to be 1-origin) as a group in a 32-bit word, ordered from least-significant bit positions to most significant. The dark shading shows data bits, the mid-shading shows control/selector bits, and the light shading shows wasted bits. Four bits are used for the selector, denoted as 's', and one bit is unused at the end of the word, denoted as 'w'. In addition, five bits are wasted internally in the sense of them being used only to store leading zeros.

Table I. Selector options used in the Simple-8b coding scheme. The *GroupSize* array is the number of items in each group, each of which is *ItemWidth* bits long.

Selector value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ItemWidth	0	0	1	2	3	4	5	6	7	8	10	12	15	20	30	60
GroupSize	240	120	60	30	20	15	12	10	8	7	6	5	4	3	2	1
Wasted bits	60	60	0	0	0	0	0	0	4	4	0	0	0	0	0	0

Anh and Moffat [10] as the Simple-9 approach because there are nine valid selector values possible for 28 data bits. Here we refer to it as being Simple-4b, to make clear that it is the 32-bit/4-byte version of the Simple approach. Note that in this case the suffix -4b refers to both the 32-bit group length and the fact that short word buffer is employed for decoding.

The situation is rather different when 64-bit units are taken as the group length. The 4-bit selector is still appropriate, leaving 60 bits available for data. But the number 60 is much more 'beautiful' (for the purposes of this style of coding, though perhaps not in other ways) than is 28, since it has many more factors. In fact, with 60 data bits, Simple-8b offers 14 different non-zero data sizes instead of the 9 that are available with 28 data bits, and of those 14, only two give rise to wasted bits. The full set of 16 selector values employed in this new Simple-8b code is listed in Table I. There are two flexible decisions in this arrangement. First, the single data bit size of 60 is rather indicative than useful. In practice, for a single text collection in a single computer node, we would expect that even the value 2^{30} is excessively large. Second, as a further extension, and to use all the possible selector values, we add two new selectors with zero data size, to handle runs of '1's in the input stream. For example, a run of 120 values $x_i = 1$ in the input would be coded into a single 64-bit word, as a 4-bit selector of value 1, followed by 60 unused bits. Zhang *et al.* [19] have also sought ways in which to exploit Simple-4b's unused selector combinations. Their proposal was to use the unused selector combinations for words of mixed code lengths, for example, using 28 bits as two 5-bit codes and three 6-bit codes. Succinctly handling long runs of '1's is probably a more useful extension and certainly keeps the decoding simpler. To justify this choice, we conducted a preliminary experiment with the document component of the inverted index for the collection .gov (described shortly in Section 6). For this case, 2.16 and 0.24% of the document gaps were coded with selector 0 (240 consecutive d-gaps all one) and 1 (120 consecutive d-gaps all one), respectively. For comparison, the median of these usage fractions over all 16 selectors was 4.34%.

The Simple-4b method potentially has many wasted bits, especially when the integers being coded are in the 10–13 and 15–25 bit ranges. The situation is clearly improved with Simple-8b, in which there are several data sizes available in these ranges. This additional flexibility should give Simple-8b better compression effectiveness than the 32-bit variant.

In terms of implementation, the critical feature that assures fast decoding is that in each group the $\lambda+1$ snips are arranged in reverse order within their machine word, as shown in Figure 2 for the case of a 32-bit group. Doing so places the selector in the least significant bits. As is the case with Simple-4b, the Simple-8b approach makes use of selectors that are absolute and are of fixed length. One mask operation is then sufficient to isolate the selector value, and with two small supporting tables, the item width θ and group size λ can be obtained efficiently, as shown in Figure 3.

In general, decoding is fast because there are no unary snips involved; because the 16 (in the Simple-8b method) different packing arrangements can be directly shift/mask programmed without looping operations being required; and because there are only $\lambda+1$ snips to be handled for every λ decoded values. In combination, these three factors mean that decoding is considerably faster than bit-by-bit decoding of bitwise codes, and is also faster than byte-by-byte decoding of Byte codes. In particular, Byte only achieves one-snip decoding for values between 0 and 127. For larger Byte values at least two snips are processed, and some additional actions must be performed to exclude the selector bit from the byte buffer before joining the latter to the decoded value. Moreover, Simple-8b is also potentially faster than Simple-4b due to the fact that the number of snips within

```
[*Successor* for Simple]
**if** $\lambda = 0$ **then**
    set $word = GetNextWord$
    set $\theta = ItemWidth[word \,\&\, 15]$
    set $\lambda = GroupSize[word \,\&\, 15]$
    set $word = word \gg 4$
**end if**
set $x = word \,\&\, mask_\theta$
set $word = word \gg \theta$
set $\lambda = \lambda - 1$
return $x$
```

Figure 3. The *Successor* operation in the Simple codes. The algorithm makes use of four coding variables: *word* for the word buffer, θ for the (current) data size, λ (initialized to 0) for the number data elements still remaining in the current buffer, and *mask* for the pre-defined array with $mask_\theta = 2^\theta - 1$. In the case of Simple-8b, the static arrays *ItemWidth* and *GroupSize* are equivalent to the second and third rows of Table I. The operation & denotes the bitwise **and** of two unsigned binary integers, and $\gg$ denotes a right-shift by the amount specified as the second operand. Note that array accesses are only made as each fresh word is commenced.

a group in the former is about as twice as that in the latter, and hence the average amortized cost of decoding selectors is roughly halved.

4.2. *The* Carry *family*

As was noted above, the Simple-4b method potentially has many wasted bits. The Carryover-12 method [10], denoted here as Carry-4b to reflect the fact that it assumes 32-bit words, is a refinement of Simple-4b designed to address that problem. In Carry-4b a selector of 2 bits is used to describe each group, and the selector of a group can be either the prefix of the group machine word or attached to the end of the previous word if there is enough free space. The selector is interpreted *relatively* to the data size of the previous group. The choice of two-bit selectors was motivated by the fact that 30 data bits give more span options than 28 does.

There are non-trivial choices to be made when extending Carry-4b to the 64-bit environment, dictated by the selector length and whether the selector interpretation is absolute or relative. To favor decoding speed, in the new Carry-8b method we chose an approach based on the ‘beauty’ of the 60 data bits used in Simple-8b, and use four-bit absolute selectors. The sole difference to Simple-8b is that, when a particular group leaves at least four wasted bits, the selector of the next group is allocated in that wasted space, leaving all 64 data bits in the next word for data. Unlike Simple-8b, two tables for selector options, one for 60 and another for 64 data bits, need to be employed.

Note that for the 60 data bits case, there are more selector options than presented in Table I for Simple-8b, arising from the desire to have space for the next selector whenever it is possible. For example, a selector for data width of 9 is now plausible, as it allows the same number of data elements as in case of 10-bit width, and also provides space for the next selector. It turns out that the number of selector options, even with the supposition that the largest data size is 28, is slightly larger than 16. We bypass this problem by employing exactly 16 consecutive options from the all-possible list, in such a way that the largest data symbol in the input stream, and nothing larger, is covered.

Compared with Simple-8b, the new Carry-8b can be expected to have better compression effectiveness, but with marginally slower decoding, because of the more complicated interpretation of selectors and the need to work with two selector tables. But note that the latter cost is small, and is associated with the group and not the individual data elements.

4.3. *The* Slide *family*

Method Slide [11], denoted here as Slide-4b, is another extension of Simple-4b, but with a critical change: a data element is permitted to span the next machine word boundary. That is, unlike the above two families, this code is not word-aligned and the word boundaries are employed simply to trigger the approximate end of groups. The selector in this approach is three-bits long, and represents a delta relative to the selector value of the previous group, rather than an absolute value. This change makes the code more effective when the magnitudes of the sequence values are locally homogeneous, but means that the selector lookup table in the decoder is two-dimensional rather than one-dimensional, which affects the decoding speed.

The Slide-4b method is readily extended to make a 64-bit Slide-8b version. The 3-bit relative selector is retained, and unused bits from one word are logically prefixed to the start of the next one, as with Slide-4b.

There are also other possible variants of Slide in a 64-bit environment. One of the possibilities is to flexibly decide the selector size based on the actual maximal element of the coded stream. This would deal with the case when the maximal value is relatively large, and the eight choices of data size offered by the 3-bit selector would lead to a non-trivial number of cases when the data size is over assigned. For example, if the maximal value is 2^{20} and the item size of the previously coded group is 7, then the eight choices for the data size of the current groups span 4 to 10, plus 20. If the current group has at least one 11-bit element, the item size is forced to be 20. With 16 choices via a 4-bit selector, these over-assigned cases are less likely. However, preliminary experiments showed that the 3-bit selector continued to be a good all-round choice, and is the version we have used in the experiments below.

More complex variants that further widen the selector, in order to support mixed-length arrangements such as five 6-bit and four 7-bit items in 58 data bits, are also less effective. It is combinatorially expensive to cover all such mixed-length scenarios, and the average gain in data bits consumed does not compensate for the increased selector cost.

With all of these three word-bounded approaches, group lengths in the 64-bit version are around double the group lengths of the 32-bit implementation, and better packing within the data part can often be achieved. That is, each coded snip requires around half of the previous selector cost, and is also more likely to contain wasted bits. But there is also a secondary effect that in tension with those gains, a possibly greater number of internal wasted bits, caused by the larger group size means that a larger diversity of data sizes is included in each group. The extent to which this is a factor depends on whether typical sequence values have locally consistent magnitudes, and is something we were interested to measure, regardless of the likely decoding speed gains.

5. FORWARDS SEEKING

As was noted in Section 2, the efficient implementation of the *Forwards Seek* operation allows fast processing of phrase and related queries such as those involving proximity operators. Seeking is also useful when the inverted index is organized in a fully interleaved fashion. In this case, data items need to be bypassed even for simple document-level *Successor* operations.

For traditional bitwise codes, unless auxiliary information is added to the index, forwards seeking operations are no faster than full decoding of the whole compressed sequence. The problem is that the unary snip for each codeword needs to be decoded first, in order to determine the (lower bound of, in case of the Golomb-8b code) length of the binary snip.

Byte codes give a level of support for *Forwards Seek* operations, because the top bit of each byte directly indicates whether or not it is the last one in the codeword. A sequential byte-scan, counting top bits, can thus be used to bypass a certain number of compressed integers, without full decoding being required on the intervening values. Nevertheless, every intervening byte must be accessed, and a mask and test operation must be applied to it.

In comparison, PackedBinary allows fast *Forwards Seek* operations, because the size of each group as well as of each item is known once the group selector has been decoded. Fast seeking is also possible in the Simple variants, as illustrated in Figure 4. Only the selector is examined in each skipped word, and because it is available in the low-order four bits, it can be processed using only a few operations. A slightly more complex arrangement is necessary for forwards seeking in the Slide code, because of the two-dimensional nature of the selectors and the fact that the selector

```
[Forwards Seek for Simple]
if $n \geq \lambda$ then
   while $n \geq \lambda$ do
      $n = n - \lambda$
      set $word$ = GetNextWord
      set $\lambda = GroupSize[word \,\&\, 15]$
   end while
   set $\theta = ItemWidth[word \,\&\, 15]$
   set $word = word \gg 4$
end if
set $word = word \gg (n \times \theta)$
set $\lambda = \lambda - n$
```

Figure 4. Forwards seeking past *n* unwanted data items using the Simple codes. All variables are as defined and used in Figure 3.

is not necessarily in the low-order bits of each word. The PForDelta-8b method can also support forwards seeking, although the need to process the exceptions at the end of each group slows it down compared with the simpler PackedBinary and the Simple methods.

The 64-bit word-bounded codes have larger groups than do the 32-bit versions. As a result, not only is decoding likely to be faster, but so too are long *Forwards Seek* operations, in which the seek distance is, in general, larger than the group size.

6. EXPERIMENTS

This section describes the experiment settings and evaluates the various claims made about 64-bit codes, using two families of randomly generated synthetic data and the index of two text collections.

6.1. Experiment settings

Two hardware systems were used in the experiments. The first system, PC32, is a 32-bit, 2.8 GHz Intel Pentium 4 with 1 GB RAM and 250 GB of local no-RAID SATA disk, running Debian Linux. The second system, PC64, is a 64-bit, quad 2.33 GHz Intel Xeon CPU with 8 GB RAM and 2 TB of no-RAID disk space, running Ubuntu Linux. All the implementations were written in `C`, and compiled with GNU `gcc` and the optimization flag '`-O3`'. All the experiments were conducted in single-thread mode, with no parallelism or concurrency employed, on otherwise idle machines.

When timing, two efficiency figures are collected: CPU and elapsed time, which are the user and real components, respectively, obtained from the Linux '`time`' command. The CPU time emphasizes in-memory decoding speed and reflects the computational cost. While more dependent on the hardware infrastructure, elapsed time reflects the real gain of index compression in terms of query processing throughput (see, for example, Ziviani *et al.* [3] and Witten *et al.* [2]). To accurately measure elapsed time, a program that cleared the whole internal memory of the system was invoked before each experimental run, to minimize the effect of the caching provided by the operating system.

As a preliminary step, we conducted experiments to validate the basic assumptions about the primitive operations on 64-bit and 32-bit architectures. In particular, it was confirmed that in the PC32 system, the same operations are substantially slower on 64-bit operands than for 32-bit ones, whereas for the PC64 system, the data size (32-bit or 64-bit) had almost no effects on the processing speed. In both the systems, the speed of the logical shift operations is practically invariant to the shift length.

6.2. Synthetic data

The initial series of experiments were conducted on two families of synthetic data. The first family, UniformData, is numbers randomly generated over the universe of nominal 'document numbers' in $1\ldots 2^{27} = 134, 217, 728$, with sets of integers generated for nominal 'terms' having document frequencies of 2^9 and 2^{18}. That is, random subsets of size 2^9 and 2^{18} of the values from $1\dots 2^{27}$ were generated, and then converted into sequences of document gaps. Multiple subsets of gaps were then concatenated until a total sequence of 400 million elements was generated, for each combination of the parameters. For example, in the 2^9 data set, there are 400 million integers, with an average gap value of $2^{27}/2^9 = 2^{18}$, and hence requiring on average an 18-bit data part.

The second family, ClusterData, has a structure similar to UniformData. However, the sequence is generated in a way that creates a clustered rather than uniform distribution, to more closely model the way that terms in an information retrieval system tend to be clustered across clumps of documents. A recursive process is used to set, in a clustered manner, f bits of the array $A[l\dots r]$ of bits, where $f \leq r-l+1$. If f is small ($f<10$) then f locations in $A[l\dots r]$ are selected, and the corresponding bits are turned on. When $f \geq 10$, the array A is randomly divided into two sub-arrays $A[l\dots m]$ and $A[m+1\dots r]$ for some choice of m, and the task becomes that of turning on $f/2$ bits in each of the sub-arrays, with care taken so that the number of 1-bits does not exceed either of the two sub-array lengths. Balancing the number of 1-bits in two (generally speaking) different-length sub-arrays achieves a sort of clustering. The three possibilities CC, CU, and UC, where C stands for clustered and U stands for uniform, are then handled via one or (in the CC case) two recursive calls to the same routine. At each such split the choice of CC, CU, and UC is made randomly, with a probability split of 0.5, 0.25, and 0.25, respectively.

Figure 5 shows the effectiveness of different coding methods for the two synthetic data sets. In each of the graphs' bars, the average compression rate attained in bits per symbol is separated into three categories: the data bits (dark shading), defined as $\sum_{i=1}^{n}\log_2 x_i$ for the sequence $x_1, x_2, \dots, x_n$ and representing the lower bound for integer coding methods; control (selector) bits, with mid-shading; and wasted (unused) bits, with light shading. The control bits are those needed for specifying the data length, for example, the bits for selectors in case of word-bounded code and the per-byte flag bit in the Byte code. All other costs—wasted leading zero bits within a binary

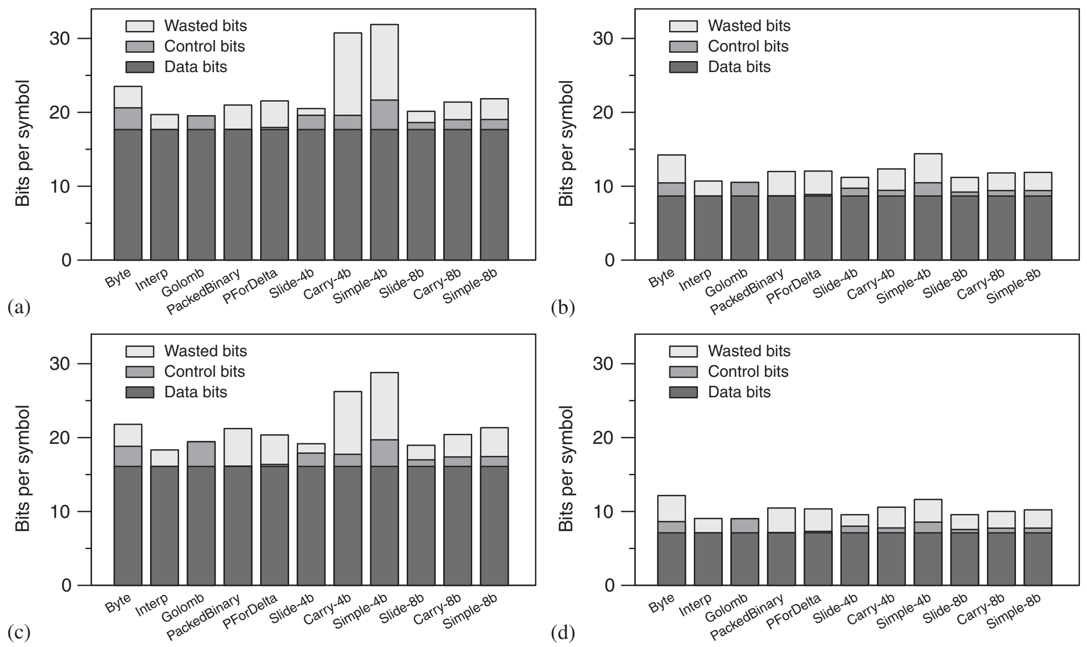

Figure 5. Compression effectiveness for four synthetic data files, representing uniform and clustered data, and rare and common terms. Each compression rate is expressed in average bits per symbol coded, as a sum of data bits (at the bottom, with dark shading), control bits (middle zone, gray shading, not always apparent), and wasted bits (at the top of each bar, light shading), where the definition of data bits depends only on the sequence values and not on the coding method. The four graphs represent: (a) uniform data, rare terms; (b) uniform data, common terms; (c) clustered data, rare terms; and (d) clustered data, common terms.

number, wasted bits at the end of a word, and so on—are combined into the third category. Note that the number of data bits is an attribute of the input sequence rather than any particular coding method.

As might be anticipated, the best compression for uniform data is attained by the Golomb code, whereas the Interpolative code Interp (see Moffat and Stuiver [23] and Cheng *et al.* [24] for refinements) is the best-performing method on the clustered data. However, for both types of data the Slide methods also work well, and there is only a small gap in the performance between the two Slide methods and the best method in each case. There is, however, a dramatic difference between Simple-4b and Simple-8b. The latter is markedly better than the former, primarily not only because of the many factors of 60, but also because we deliberately chose a test sequence (in parts (a) and (c) of Figure 5) with an average symbol value that would push Simple-4b to bad performance. The performance of the Carry methods falls between the corresponding Slide and Simple methods, but is closer to the latter. In all of the cases shown in Figure 5, the Byte code is inferior in comparison with the bitwise and Slide codes, and although it is better than Simple-4b for big gaps, it is inferior to Simple-8b in all cases.

Table II compares the decoding speed of several methods using the ClusterData on each of the hardware systems. Both CPU and elapsed time are reported, in nanoseconds per decoded symbol, averaged over a sequence of 400 million symbols. Note that the suffixes on the code names describe the length of the word buffer employed in decoding.

A number of observations arise from Table II:

- There is a dramatic difference between the CPU and the elapsed time, which suggests that in-memory techniques, like list caching [19], are crucial to improve the overall processing rates. The choice of coding method is then especially important for systems that facilitate in-memory inverted lists.
- Even when data is being processed from disk, the fact that CPU cost is primarily affected by algorithmic issues, whereas I/O cost is primarily (but not exclusively) affected by data volumes means that there are non-trivial tradeoffs that affect the overall elapsed time.
- Bitwise codes require much more CPU effort than the others in terms of decoding speed, and the interpolative code is slower still. Even when measured by elapsed time, these two are notably slower than the others.

Table II. Average decoding time for the data sets of the ClusterData family, using the two test architectures, and measured over sequences of 400 million values. All CPU and elapsed times are measured in nanoseconds per symbol.

	Average CPU decoding time				Average elapsed decoding time			
	Rare term		Common term		Rare term		Common term	
Method	PC32	PC64	PC32	PC64	PC32	PC64	PC32	PC64
Byte	8.37	5.06	6.38	3.76	52.19	39.51	30.96	23.08
Golomb-4b	30.83	14.38	26.36	13.06	70.87	44.98	41.70	27.20
PForDelta-4b	7.09	3.50	7.24	2.94	49.07	36.44	27.65	18.72
PackedBinary-4b	6.58	3.32	6.31	2.70	48.45	36.15	27.49	19.02
Slide-4b	10.29	8.02	8.68	5.92	51.26	38.18	28.12	20.94
Carry-4b	8.71	4.68	6.25	3.50	61.39	39.28	28.88	20.12
Simple-4b	9.45	4.33	6.20	3.16	67.11	38.27	30.44	19.55
Interp-8b	124.84	35.63	114.81	31.60	161.81	64.56	133.33	45.96
Golomb-8b	33.65	15.32	27.62	12.34	72.94	46.00	44.78	26.60
PForDelta-8b	10.86	3.18	9.71	2.74	52.88	36.14	29.96	18.59
PackedBinary-8b	10.06	2.72	8.89	2.42	52.22	35.71	29.88	18.78
Slide-8b	12.87	5.70	10.23	4.56	52.14	35.68	28.57	19.70
Carry-8b	9.41	3.78	6.88	2.66	50.71	35.70	27.29	18.49
Simple-8b	9.32	3.44	6.52	2.46	52.33	35.88	27.63	18.43

Table III. Compression performance for the four component sequences extracted from the inverted index of the .gov text collection.

	Average bits per symbol by categories				Total size		
						As percentage of	
Method	Document gaps	Position gaps	Term frequencies	Term impacts	GB	Interp	Simple-8b
Interp	5.81	9.01	2.44	1.92	2.01	100	84
Golomb	6.03	9.57	2.80	2.09	2.14	106	90
Byte	9.37	11.16	8.02	8.00	3.06	152	128
PForDelta	7.17	10.80	4.74	3.50	2.55	127	107
PackedBinary	7.30	11.99	5.43	2.35	2.74	136	115
Slide-8b	6.44	9.92	4.02	2.15	2.28	113	95
Carry-8b	6.66	10.22	4.00	2.26	2.34	116	98
Simple-8b	6.82	10.45	4.02	2.26	2.39	119	100

The first four data columns are measured as bits per symbol for each separate component. The next column represents the total index size, summing all four components, whereas the last two columns express the total size relative to the sizes of the Interp and Simple-8b indexes.

- There are modest but definite differences in the average CPU speed between the other coding methods.
- Use of a long buffer word gives consistently faster decoding on the 64-bit machine. For the 32-bit system, the use of long word operations slows down the decoding speed.
- If compression effectiveness is also factored into the equation, then Simple-8b is perhaps a slightly better combination than other codes for the 64-bit systems, and the choice for the 32-bit systems is probably PForDelta-4b or even PackedBinary-4b. However, it is also worth noting that for 32-bit systems it is possible to employ Simple-8b (and Carry-8b, Slide-8b) with a short-word decoding buffer, a flexibility that is not possible with the other codes. We will return to this point again in Section 7.

6.3. Text collection index

In order to also provide results on 'real' data, a set of test sequences was constructed from the 18 GB TREC text corpus .gov (see http://trec.nist.gov), a document collection containing approximately 1.25 million web documents crawled from the .gov domain. The inverted index for this collection was divided into four separate test files: the document number gaps, the word position number gaps, the document frequencies, and integer document impacts used in computing the similarity score between each document and the query. The set of position gaps has around 1.5 billion elements, and each of the other sets has approximately 380 million elements.

Table III shows the compression effectiveness achieved on each of the four symbol streams, using a range of methods. The Golomb and Interp methods are again superior if only compression effectiveness is taken into account, and Slide-8b remains the best among the others. The Byte code obtains relatively poor effectiveness on the term frequencies and term impacts.

6.4. Text retrieval performance

To carry out a 'live' comparison, a subset of these coding methods was then embedded into a research retrieval system. The methods tested included Byte, as a baseline; and Bitwise, PForDelta-8b, Simple-8b, and Slide-8b. The Bitwise approach uses Golomb-8b codes for document gaps, Delta codes for the term positions, and Gamma codes for the term frequencies and impacts—in essence, the combination proposed by Witten *et al.* [2], and also used as a reference point in the experiments of Scholer *et al.* [4].

All testing was carried out using the 426 GB .gov2 document collection, also provided as a part of the TREC resources, which contains around 25 million web documents from the .gov domain. Two query sets were employed. The first set contained the first 10 000 queries of the 100 000

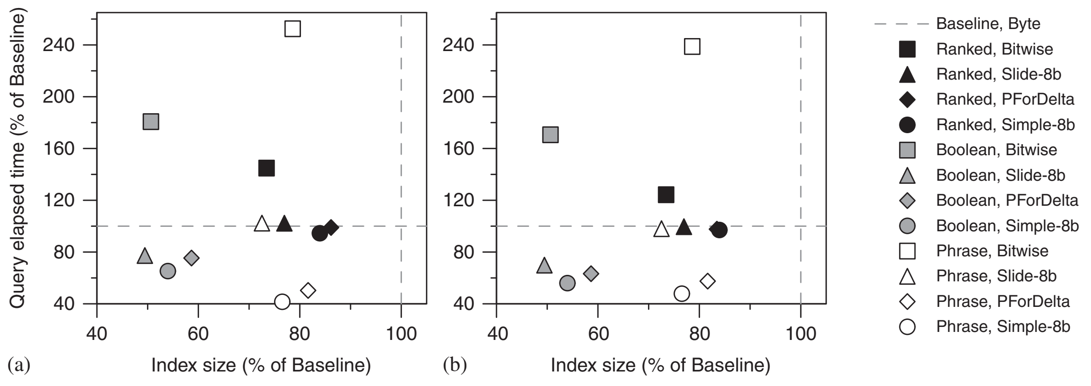

Figure 6. Relative trade-off between index space and query processing time for five coding methods, over three different query types. All size and speed results are normalized percentages relative to the size and the speed of that type of query when executed using a Byte coded index. Two different architectures were used: (a) a 32-bit machine and (b) a 64-bit machine. For the ranked and the Boolean queries, 100 000 queries were used; for the phrase queries, 3395 queries were used.

Table IV. Reference points for Byte performance, for three different query modes, three corresponding inverted file structures, and two different architectures. The points plotted in Figure 6 are normalized using these three sets of values.

Query mode	Space (% text)	Time (s/q) PC32	Time (s/q) PC64
Ranked	1.9	0.30	0.25
Boolean	3.2	0.25	0.18
Phrase	12.7	1.26	1.08

queries used in the Terabyte track of TREC-2006. The second set was built from the first by extracting the queries that contained at least two parsed terms and had at least one .gov2 answer when considered as a phrase query. The average query length in the first set was 4.11 words, and the average and the median number of conjunctive Boolean answers were 37 609 and 2202, respectively. The second set had 3395 queries of average length 3.15 words, and an average and median number of phrase answers of 2620 and 17.

The performance of the various index codes were explored in three different query environments: ranked queries with impact-sorted indexes, where the index fits the need of query processing, with no redundancy; Boolean conjunctive queries using a document-sorted index containing document gaps and term impacts (an index usable for both Boolean and ranked queries); and phrase queries using an 'all four components' index. The results are shown in Figure 6, in which the performance of Byte is taken as a '100% baseline' for space and speed on the two architectures, for each of the three different query types. The nine normalization factors used are given in Table IV.

In terms of space, the difference between the word-bounded and the Bitwise regime is modest, with the Simple-8b code (circular marks in the graphs) only slightly more expensive than Slide-8b (triangular marks) in the three index structures explored, and the Bitwise code (square marks) giving a clear advantage only on the index used for the ranked queries (black marks). In terms of query speed, the new Simple-8b method provides the fastest relative performance, and the circular marks corresponding to Simple-8b are consistently the lowest in each grouping across the two graphs (the dark-shaded group shows ranked queries; the mid-shading shows Boolean queries; and white shading shows phrase queries). Of the methods tested, only the Bitwise implementation (square marks) is slower than the Byte approach (the dotted reference lines), whereas

PForDelta-8b (diamond marks) also performs well; it is slightly inferior in terms of both efficiency and effectiveness than the Simple-8b code (circles).

Note how the choice of index compression regime is less critical for ranked queries (as evidenced by the small vertical spread of the dark-shaded marks) than it is for Boolean and phrase queries—the latter two being the modes in which *Forwards Seek* is an important component. Note also that the same pattern of relative performance is in evidence for both architectures, and it is apparent that we can safely make use of 64-bit codes on 32-bit architectures to get good compression effectiveness and fast decoding.

7. BULK UNPACKING

The previous experiments focussed on the use of 32-bit versus 64-bit words as constraints on the code, and also as the processing buffer for decoding. Moreover, the compressed stream was decoded in a value-by-value manner: each individual codeword was decoded only when actually needed. This decoding interface facilitates within-block *Forwards Seek* operations, and provides a generic interface suited to a wide range of applications. The Interp and PForDelta codes, however, do not directly support value-by-value decoding because of the dependencies that exist between elementary codewords, and they were implemented using a block-by-block approach. Indeed, the speed of the PForDelta code is in no small part a consequence of the fast bulk unpacking methods described by Zukowski *et al.* [18] and Zhang *et al.* [19]. Their procedure extracts all of the same-size elements in each group using a linear sequence of primitive operations, without relying on any loop or conditional jumps.

A similarly focussed unpacking regime can also be applied to the PackedBinary, Simple, and Carry code families. For Simple-8b, bulk decoding gives a significant speedup, especially for small data sizes. For larger data sizes, each Simple-8b block contains relatively few elements, and the effect of bulk decoding is less dramatic. Even so, Simple-8b should still be reasonably fast, because there is no need to handle the exceptions that PForDelta-8b has to deal with. To test this belief, we implemented block-by-block bulk decoding for Simple, Carry, PForDelta, and PackedBinary, for the two hardware systems described earlier. In addition, we also implemented Simple-8-4b which is essentially Simple-8b in terms of the code used, but uses a short 32-bit buffer word for decoding operations, with one selector extracted for every second of word processed. Note that although the straight-line programs implementing bulk decoding are neither elegant nor examples of good software engineering, they can be generated in a straightforward manner by simple scripts.

One drawback of bulk unpacking is that each group of same-size binary codewords should be word-aligned, with byte-alignment not an option because of the implicit alignment of the hardware. The word-alignment does not add any additional space cost to the word-bounded codes, but it does add a non-trivial cost to PForDelta and PackedBinary, since it means a substantial waste after the codeword of parameters, as well as after the codewords of exceptions. This cost, in terms of compression effectiveness, can be estimated based on the word size used.

The results for these implementations are shown in Table V, summed over the four components of the index of the .gov collection. Note that because of the word alignment requirements, the PForDelta and PackedBinary codes are slightly changed, and the suffixes -4ba and -8ba reflect the length of the decoding buffer (as before), and additionally that each of the sub-lists is word-aligned. Overall, the figure shows that the 64-bit versions of word-bounded codes have clear compression advantages, but that the PForDelta variants maintain a slight speed advantage.

A comparison between Table V and the figures reported in the recent literature shows that the decoding speeds shown for Byte in PC64 are similar to those of Trotman and Subramanya [21], indicating that the implementations had a comparable level of tuning put into them. The absolute decoding speed attained by our PForDelta-8b is also comparable with that reported by Zhang *et al.* [19]. To allow others to similarly build on our work, the implementations of the Byte, PForDelta and word-bounded codes reported here are available at http://www.csse.unimelb.edu.au/∼alistair/coders-64bit.

Table V. Performance of Byte, as a reference point, and a range of coding methods when linear-sequence bulk unpacking is employed in decoding.

Method	Average bits per symbol	Average decoding time: CPU PC32	CPU PC64	Elapsed PC32	Elapsed PC64
Byte	9.99	5.19	2.99	25.50	16.87
PackedBinary-4ba	9.17	1.42	1.11	20.04	15.54
PForDelta-4ba	8.45	2.31	1.46	19.73	14.80
Carry-4b	7.93	4.70	2.67	20.72	15.23
Simple-4b	8.25	3.32	2.18	20.04	14.74
PackedBinary-8ba	9.42	3.92	0.97	23.06	15.86
PForDelta-8ba	8.82	5.00	1.49	23.16	15.43
Carry-8b	7.65	5.05	1.72	20.59	13.79
Simple-8b	7.81	4.45	1.44	20.36	13.88
Simple-8-4b	7.81	2.13	1.53	18.21	14.15

The index is for .gov collection, with all four components of each posting represented as separate streams. The average decoding time is measured in nanoseconds per symbol, assuming that the full index is being decoded, but not needing to be processed or written.

8. SUMMARY AND REMARKS

We have explored the option of using 64-bit units when designing word-bounded codes for index compression, as well as using 64-bit word buffers when decoding these codes. We have shown that for the now-dominant 64-bit computer systems, the use of long words offers compression and/or speed advantages. Word-bounded codes benefit from the use of 64-bit words in setting the group boundaries, and on 32-bit architectures the codes can still be accessed using a 32-bit decoding buffer. In this regard our results complement those of Büttcher and Clarke [16] and Zhang *et al.* [19], who have also explored the tradeoffs involved with the inverted index compression, including the evaluation of the 32-bit word-bounded Simple-4b code.

In addition, the Simple-8b and Simple-4b codes possess a property that none of the others do: codewords are never split over word boundaries, and the values coded in each machine word can be decoded without relying on any external information. This property is not relevant to the text retrieval context used in the experiments reported here, but might be helpful when word-by-word communication is needed. More generally, Simple-8b can be applied to any situation in which streams of integers are to be represented, and can be expected to provide both useful compression effectiveness and fast decoding.

ACKNOWLEDGEMENTS

This work was supported by the Australian Research Council and the ARC Center for Perceptive and Intelligent Machines in Complex Environments, and by a Google Research Awards grant.

REFERENCES

1. Zobel J, Moffat A. Adding compression to a full-text retrieval system. *Software—Practice and Experience* 1995; **25**(8):891–903.
2. Witten IH, Moffat A, Bell TC. *Managing Gigabytes*: *Compressing and Indexing Documents and Images* (2nd edn). Morgan Kaufmann: San Francisco, 1999.
3. Ziviani N, de Moura E, Navwrro G, Baeza-Yates R. Compression: A key for next-generation text retrieval systems. *Computer* 2000; **33**(11):37–44.
4. Scholer F, Williams H, Yiannis J, Zobel J. Compression of inverted indexes for fast query evaluation. *Proceedings of 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, Beaulieu M, Baeza-Yates R, Myaeng S (eds.). ACM Press: New York, August 2002; 222–229.

5. Trotman A. Compressing inverted files. *Information Retrieval* 2003; **6**(1):5–19.
6. Golomb S. Run-length encoding. *IEEE Transactions on Information Theory* 1966; **12**(3):399–401.
7. Zobel J, Moffat A. Inverted files for text search engines. *ACM Computing Surveys* 2006; **38**(2):1–56.
8. Manning CD, Raghavan P, Schütze H. *Introduction to Information Retrieval*. Cambridge University Press: Cambridge, 2008. Web site accessed [January 2008].
9. Büttcher S, Clarke CLA. Unaligned binary codes for index compression in schema-independent text retrieval systems. *Technical Report*, University of Waterloo, 2006; 10.
10. Anh VN, Moffat A. Inverted index compression using word-aligned binary codes. *Information Retrieval* 2005; **8**(1):151–166. Source code available from: www.cs.mu.oz.au/~alistair/carry/ [October 2009].
11. Anh VN, Moffat A. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering* 2006; **18**(6):857–861.
12. Strohman T, Turtle H, Croft WB. Optimization strategies for complex queries. *Proceedings of 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, NY, U.S.A., Marchionini G, Moffat A, Tait J, Baeza-Yates R, Ziviani N (eds.). ACM Press: New York, August 2005; 219–225.
13. Barbay J, López-Ortiz A, Lu T. Faster adaptive set intersections for text searching. *Proceedings of Fifth International Workshop on Experimental Algorithms* (*Lecture Notes in Computer Science*, vol. 4007), Àlvarez C, Serna MJ (eds.). Springer: Berlin, May 2006; 146–157.
14. Moffat A, Zobel J. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems* 1996; **14**(4):349–379.
15. Strohman T, Croft WB. Efficient document retrieval in main memory. *Proceedings of 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Amsterdam, The Netherlands, Clarke CLA, Fuhr N, Kando N, Kraaij W, de Vries AP (eds.). ACM Press: New York, July 2007; 175–182.
16. Büttcher S, Clarke CLA. Index compression is good, especially for random access. *Proceedings of 2007 ACM CIKM International Conference on Information and Knowledge Management*, Lisbon, Portugal, Silva MJ, Laender AHF, Baeza-Yates RA, McGuinness DL, Olstad B, Olsen ØH, Falcão AO (eds.). ACM Press: New York, November 2007; 761–770.
17. Culpepper JS, Moffat A. Compact set representation for information retrieval. *Proceedings of 14th International Symposium on String Processing and Information Retrieval* (*Lecture Notes in Computer Science*, vol. 4726), Santiago, Chile, Ziviani N, Baeza-Yates R (eds.). Springer: Berlin, October 2007; 137–148.
18. Zukowski M, Heman S, Nes N, Boncz P. Super-scalar RAM-CPU cache compression. *Proceedings of the 22nd International Conference on Data Engineering*, Altanta, GA, Liu L, Reuter A, Whang K-Y, Zhang J (eds.). IEEE Computer Society: Silver Spring, MD, April 2006; 59.
19. Zhang J, Long X, Suel T. Performance of compressed inverted list caching in search engines. *Proceeedings of 17th Conference on the World Wide Web*, Beijing, China, Huai J, Chen R, Hon H-W, Liu Y, Ma W-Y, Tomkins A, Zhang X (eds.). ACM Press: New York, April 2008; 387–396.
20. Brisaboa N, Fariña A, Navarro G, Esteller M. (S, C)-dense coding: An optimized compression code for natural language text databases. *Proceedings of Symposium on String Processing and Information Retrieval* (*Lecture Notes in Computer Science*, vol. 2857), Manaus, Brazil, Nascimento MA (ed.). Springer: Berlin, October 2003; 122–136.
21. Trotman A, Subramanya V. Sigma encoded inverted files. *Proceedings of 2007 ACM SIGIR International Conference on Information and Knowledge Management*, Silva M, Laender A, Baeza-Yates R, McGuinness D, Olstad B, Olsen O, Falcao A (eds.). ACM Press: New York, 2007; 983–986.
22. Williams HE, Zobel J. Compressing integers for fast file access. *The Computer Journal* 1999; **42**(3):193–201.
23. Moffat A, Stuiver L. Binary interpolative coding for effective index compression. *Information Retrieval* 2000; **3**(1):25–47.
24. Cheng CS, Shann JJJ, Chung CP. Unique order interpolative coding for fast querying and space-efficient indexing in information retrieval systems. *Information Processing and Management* 2006; **42**(2):407–428.