Hybrid Compression of Bitvectors for the FM-Index[*]

Juha Kärkkäinen Dominik Kempa Simon J. Puglisi

Helsinki Institute for Information Technology (HIIT) and
Department of Computer Science, University of Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

Abstract

Compressed bitvectors supporting rank and select operations are the workhorse of compressed data structures. We propose a hybrid scheme for implementing compressed bitvectors, which divides the bitvector into blocks and then chooses the encoding of each block separately from a number of different encoding methods. Hybrid encoding is particularly suitable for bitvectors that have lots of local and regional variation, such as those present in the FM-index, a popular compressed data structure for pattern matching. We propose a specific hybrid combination of three simple encoding methods for FM-index bitvectors achieving superior space-time tradeoffs in experiments.

1 Introduction

Compressed data structures store data in compressed form while supporting a set of nontrivial operations on the data. They can be more than an order of magnitude smaller than traditional data structures. Perhaps the biggest success story among compressed data structures is the FM-index [4], a compressed full-text self-index [17] that stores a string and supports pattern matching queries. FM-indexes are critical to several widely used bioinformatics tools (see, e.g., [6, 12]).

A common principle in compressed data structures is to reduce complex operations on complex data into simpler operations on simpler data. A prime example is the FM-index with two reduction steps: (1) The *Burrows–Wheeler transform* (BWT) [1] is an invertible permutation of the text that turns a pattern matching query on the text into a sequence of *rank* queries on the BWT; and (2) The *wavelet tree* [8] is a representation of the BWT as a set of bitvectors that turns a BWT rank query into a sequence of rank queries on bitvectors. As in the case above, at the end of the chain of reductions is often a bitvector supporting rank and select operations ($b \in \{0,1\}$):

$$\begin{aligned}\mathrm{rank}_b(i) &= \text{number of } b\text{s among the first } i \text{ bits}\\ \mathrm{select}_b(j) &= \text{position of the } j\text{th } b\end{aligned}$$

[*]Partially supported by Academy of Finland grants 118653 (ALGODAN) and 250345 (CoECGR).

A single type of compressed bitvector is unlikely to be sufficient for all purposes as the types of queries and the compressibility of the bitvectors varies from application to application. FM-indexes primarily use rank queries (select is needed rarely) and so it follows that FM-index bitvectors should be optimized for rank queries.

The compressibility of FM-index bitvectors is rather complicated. Several variants of the FM-index can achieve a space complexity close to nH_k, where n is the length of the text and H_k is the kth order empirical entropy [16] of the text. This is possible even with uncompressed bitvectors. A basic zero-order entropy compression is achieved by using a Huffman-shaped wavelet tree [9]. For higher order entropy compression, we can use a technique called *compression boosting* [2, 5, 11], where the BWT is partitioned into blocks and each block is zero-order compressed. On the other hand, using bitvectors compressed with the method known as RRR [21], kth order entropy compression is achieved without any compression at the higher levels [14]. RRR partitions the bitvectors into small blocks and compresses each block separately close to zero-order entropy, which achieves *implicit compression boosting* [14] when applied to a balanced wavelet tree of the BWT.

Even higher-order entropy compression is not sufficient for compressing highly repetitive data sets such as web crawls, whole genome databases, and versioned repositories. The long repeating sections in such data are not captured by H_k for any reasonable k but manifest as runs in the BWT, and run length encoding has been successfully used as the method of compression. Again, the compression can be applied directly to the BWT [13] or to the bitvectors [15].

The variable nature of the FM-index bitvectors makes it difficult to find a good compression method for them. Such a method should compress well regions with low zero-order entropy as well as regions with long runs, but without blowing up the size of less compressible regions. At the same time it should support fast rank operations. All current approaches have a weakness in some area. RRR achieves inoptimal compression of highly repetitive data with long runs, while run length encoding performs poorly for less repetitive data. Neither method matches uncompressed bitvectors in query speed.

Our Contribution We propose a new technique for implementing compressed bitvectors based on combining multiple basic compression methods. The bitvector is partitioned into blocks and the encoding of the block is chosen separately for each block. We call this *hybrid compression.* The advantage is that we can design each individual compression method to perform well for a certain type of blocks without caring how it performs for other types.

A minor disadvantage of hybrid compression is a small overhead in time and space because of the selection of the compression method for each block. However, the cost of a few extra conditional branches is insignificant compared to the cost of cache misses, which can dominate the query time in practice, and the cost of some extra space per block can be reduced by using larger blocks.

We have designed and implemented a simple hybrid compression for FM-index bitvectors consisting of three encoding methods: simple run length encoding for blocks with a small number of runs, simple minority bit position encoding for blocks with a small zero-order entropy, and plain (uncompressed) encoding for other blocks. In experiments with several types of data, an FM-index with the new hybrid bitvector displays excellent all-around performance when compared against other FM-index implementations. In all cases, it is among the best compressing and among the fastest. Every other implementation performs poorly on at least one data set. In several cases, the new FM-index achieves space-time tradeoffs unmatched by any other method.

Related Work As they are so central to succinct and compressed data structures, many methods for supporting rank and select over bitvectors have been proposed. These methods are generally of two types: uncompressed and compressed. Uncompressed methods keep the original bitvector intact, and build sublinear data structures over it to support the desired functions. The best uncompressed solutions are due to Navarro and Providel [18], Zhou et al. [22] and Gog and Petri [7].

Compressed solutions on the other hand replace the original bitvector with some compressed representation which supports rank and select queries over the original. The most widely used compressed solution is the so-called RRR encoding by Raman, Raman and Rao [21], which requires $H_0 + o(n)$ bits of space and supports rank and select operations in constant time. The most efficient implementations of RRR are recent and due to Navarro and Providel [18] and Gog and Petri [7]. Sadakane and Okanohara [20] describe several different compressed representations particularly for sparse bitvectors. Gupta et al. [10] describe a compressed encoding with space close to the *gap* measure, which can be much smaller than H_0 when the bits are unevenly distributed. Mäkinen et al. [15] describe how bitvectors with many runs can be encoded using sparse bitvectors and suggest using Gupta et al.'s encoding for those.

There are many experimental comparisons of different variants of FM-indexes and other compressed text indexes; the most recent one is by Gog and Petri [7]. Mäkinen et al. [15] focus on run-length compressed indexes. The best index in their comparison is the run-length compressed suffix array (RLCSA), which is not an FM-index. Nevertheless, we include RLCSA in our experiments as it outperforms run-length compressed FM-indexes in many cases.

The idea of choosing between multiple bitvector encodings in a wavelet tree has been suggested by Navarro et al. [19] though at a higher level and in a different context. In a wavelet tree of a document array, they choose the encoding separately for each level of the wavelet tree.

2 Hybrid Bitvector for the FM-Index

In this section, we describe a hybrid bitvector designed for the FM index at a general level. The implementation used in the experiments is described in the next section.

The bitvector $B[0..n)$ is divided into blocks of size b: $B = B_0 B_1 \dots B_{n/b-1}$, where $B_i = B[ib..(i+1)b)$ (for simplicity we assume that b divides n). The primary representation of a block B_i consists of two parts, a fixed-size header and a variable-size body. The header contains (1) the *weight* of the block (i.e., the number of ones) (2) the size of the body, and (3) additional information about the encoding method of the body. The body contains sufficient information so that, together with the header information, the block can be reconstructed.

The blocks are grouped into superblocks, each consisting of b_s blocks. For each superblock, we store a fixed-size header containing the cumulative sums of (1) block weights and (2) body sizes from the beginning of B to the beginning of the superblock. The former cumulative sum tells the rank value at the beginning of the superblock and the latter tells the location of the body of the first block in the superblock. The block and superblock headers are stored in a single array R_h interleaved so that each superblock header is followed by the block headers of that superblock.

We use three different types of encodings for the bodies:

1. Plain bitvector of b bits.

2. Minority bit positions: the positions of all zero bits or all one bits, whichever has fewer occurrences, using $\lceil \log b \rceil$ bits per minority bit. The number and type of the minority bits can be determined from the header information.

3. Run length encoding: the lengths of runs of zeros and ones using $\lceil \log b \rceil$ bits for each run. The size of the body stored in the header tells the number of runs, and one bit in the header tells whether the first run is zeros or ones.

The type of encoding is chosen to minimize the size of the body. The bodies, which are of variable length, are concatenated together into an array R_b.

To answer a query $\mathrm{rank}_1(i)$, we lookup the superblock header containing i and scan the block headers from the beginning of the superblock up to the block B_j that contains the position i. Using the information collected, we can compute the rank at the beginning of B_j and locate the body of B_j in R_b. We then scan the body to answer the query. Note that $\mathrm{rank}_0(i) = i - \mathrm{rank}_1(i)$.

Several optimizations of this scheme are possible:

- Each block header stores the cumulative sums of weights and body sizes from the beginning of the superblock. This way we can avoid scanning the block headers and access the relevant header directly.

- By rounding the plain encoding size up to a multiple of $\lceil \log b \rceil$ bits, the size of a body is always a multiple of $\lceil \log b \rceil$ bits. Then we need only $\lceil \log(b/\lceil \log b \rceil) \rceil$ bits to store a body size.

- The type of the encoding can often be determined from the weight and the body size. In fact, the only additional information necessary in the header is one bit to tell the type of the first run for run length encoding.

- In the run length encoding, the lengths of the last two runs are not stored since they can be determined from the lengths of the other runs and the header information. In particular, if the block has at most two runs, the body is empty.

- For any of the encodings and any constant $\epsilon > 0$, we can construct a lookup table of size $\mathcal{O}(n^\epsilon)$ that allows scanning the body $\Theta(\log n)$ bits at a time.

With these optimizations, a rank query can be answered in $\mathcal{O}(1 + b/\log n)$ time. The space needed for the headers is

$$\frac{n}{b}\left(\lceil\log(b_s b - b + 1)\rceil + \left\lceil\log\frac{b_s b - b + 1}{\lceil\log b\rceil}\right\rceil + 1\right) + \frac{n}{b_s b}\left(\lceil\log n\rceil + \left\lceil\log\frac{n}{\lceil\log b\rceil}\right\rceil\right)$$

If we choose $b_s, b = \Theta(\log n)$, the rank query time is constant and the total size of the headers is $\mathcal{O}(n \log\log n/\log n)$. Since the size of a block body is never more than $b + \log b$ bits, the total size of the data structure is never more than $n + \mathcal{O}(n\log\log n/\log n) = n + o(n)$ bits. Similarly, if B has r runs and m minority bits, the data structure uses no more than $\min\{r, m\}\lceil\log b\rceil + o(n)$ bits.

The hybrid bitvector has a blind spot at slightly compressible blocks. For example, if a block has about $b/\log b$ minority bits and at least as many runs, none of the three techniques achieves compression, but RRR with the same block size could compress the block to $\mathcal{O}(b\log\log b/\log b)$ bits. This weakness could be addressed by adding more encodings into the mix to handle such blind spots. However, we have chosen not to do so in order to keep the implementation simple and very fast.

The hybrid bitvector can support select operations too, but locating the relevant superblock and block requires either binary searching in additional $\mathcal{O}(\log n)$ time, or the use of additional data structures.

3 Implementation

We have implemented a restricted version of the hybrid bitvector of the previous section. The emphasis in the implementation is simplicity and speed rather than squeezing out every last bit. The key implementation decisions are:

- Block headers store weights and body sizes rather than their cumulative sums, so scanning of the block headers is required. Scanning is a very fast operation in practice, because it does not involve cache misses. This also leads to a convenient header size of two bytes (see below).

- We fix $b = 255$. All block weights, minority bit positions and run lengths are encoded with one byte. Body sizes are expressed in bytes using six bits. Plain encoding uses 32 bytes, i.e., 256 bits.

- The block header size is 16 bits consisting of block weight (8 bits), body size (6 bits), and two bits for the type of block encoding. The four encoding types are plain encoding, minority bit positions, run length encoding starting with a run of ones, and run length encoding starting with a run of zeros.

- The superblock header size is 64 bits containing the sum of weights (32 bits), the sum of body sizes (29 bits), and one bit for the special case of a superblock containing only ones or only zeros. The number b_s of blocks in a superblock is a parameter that varies between 8 and 64.

Name	σ	$n/2^{20}$	n/r	n/z	Source	Description
dna	16	200	1.63	15.0	S	Human genome
proteins	25	200	1.93	8.5	S	Swissprot database
english	225	200	2.91	15.0	S	Gutenberg Project
sources	230	200	4.40	18.3	S	Linux and GCC sources
dblp.xml	96	200	7.09	29.9	S	DBLP bibliography
para	5	410	27	184	R	S. Paradoxus genome
cere	5	440	40	272	R	S. Cerevisae genome
influenza	15	148	51	199	R	Influenza DNA
world_leaders	89	44	82	267	R	CIA World Leaders
kernel	160	246	93	324	R	Linux Kernel sources

Table 1: Files used in the experiments. The files are from (S) the Pizza & Chili standard corpus (`http://pizzachili.dcc.uchile.cl/texts.html`) and (R) the Pizza & Chili repetitive corpus (`http://pizzachili.dcc.uchile.cl/repcorpus.html`). The values of n/r (average length of run in BWT) and n/z (average length of phrase in LZ77 factorization) are included as measures of repetitiveness.

- We do not use lookup tables when processing block bodies. Minority bit positions and run-length encoding are scanned byte by byte. Plain encoding is scanned using 64-bit built-in popcount instructions, which is the fastest method according to Gog and Petri [7].

The total size of this encoding in the worst case, i.e., when all blocks are encoded as plain bitvectors, is between $1.07n$ and $1.1n$ bits depending on the value of b_s. The overhead of 7–10% is the total size of the headers and represents a lower bound below which the bitvector cannot be compressed.

4 Experiments

Setup. We performed experiments on a 3.4GHz Intel Core i7-3770 CPU equipped with 8MB L3 cache and 16GiB of DDR3 main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.2.0. All programs were compiled using `g++` version 4.7.3 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. All given runtimes are wallclock times, recorded with the Unix `gettimeofday` function.

Methodology. We reproduce the experiments by Ferragina et al. [3], and Gog and Petri [7] measuring the performance of count queries. A count query, which returns the number of occurrences of the pattern, is the most fundamental operation in an FM-index (and RLCSA) and the most dependent on the bitvector rank operation. For each file in our data set (detailed in Table 1) we extracted 5×10^4 patterns of length 20 at random positions and use them as queries.

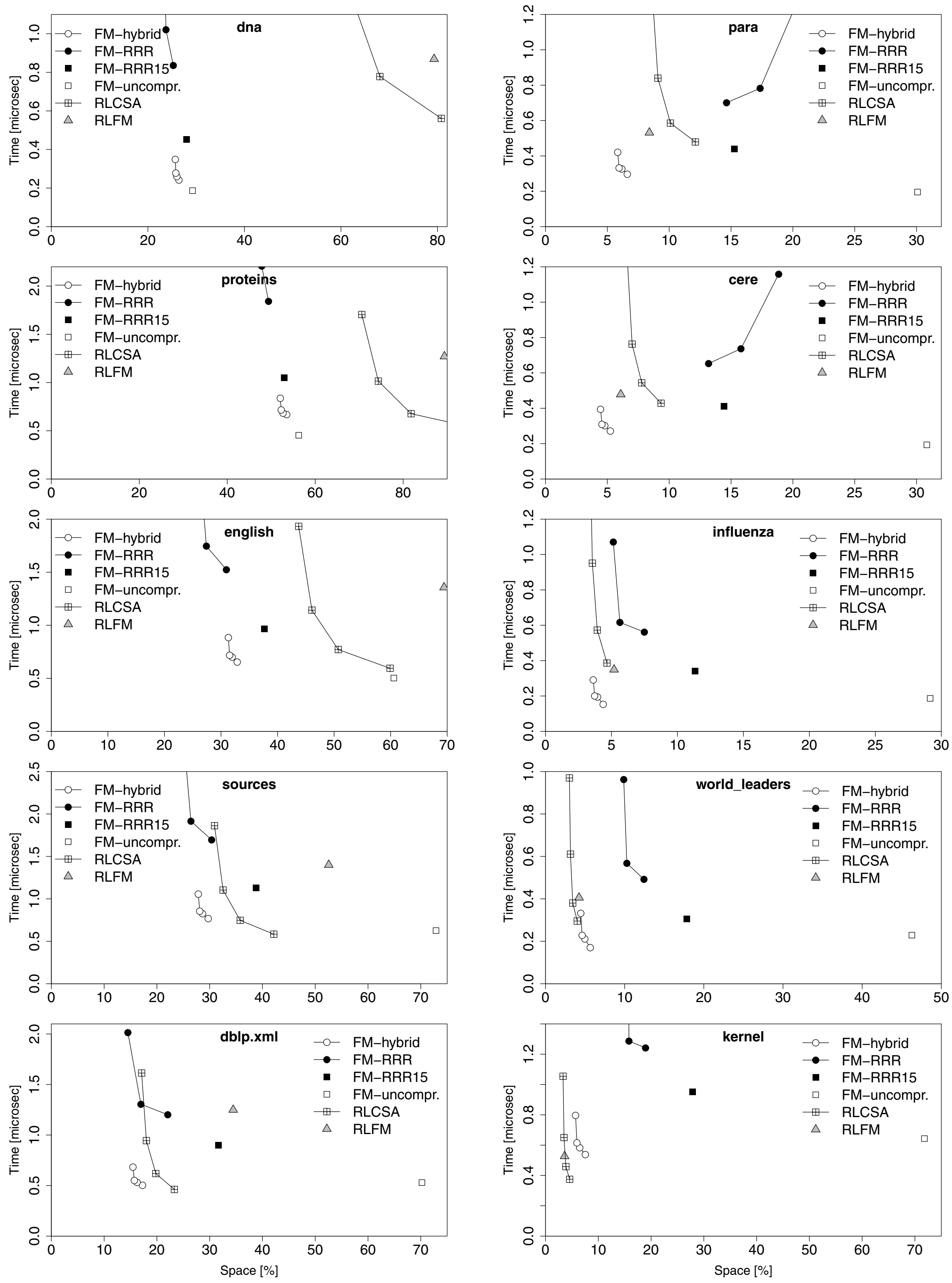


Figure 1: Time/space tradeoffs for *count* queries on standard (left) and repetitive (right) data sets. The time is the average wallclock time (in μsec) to match one symbol during a single count query. Times were obtained by taking an average over 5×10^4 queries on 20-length patterns randomly extracted from the indexed text. Space is given with respect to the original size of the input text.

The text indexes included in the experiments are:

- FM-RRR is a Huffman-shaped wavelet tree of the BWT of the text with the wavelet tree bitvectors encoded by RRR implemented without lookup tables as described in [18, 7]. The block sizes are 31, 63, and 127.
- FM-RRR15 is the same as above but using a highly optimized implementation of RRR with block size 15.
- FM-uncompressed is the same as above but using highly optimized uncompressed bitvectors called V5 in [7].
- FM-hybrid is the same as above but with the wavelet tree bitvectors encoded by our new hybrid encoding method. The superblock sizes are 8, 16, 32 and 64.
- RLFM is an FM-index based on a run-length encoded BWT.
- RLCSA is a run-length encoded index described in [15]. It is not an FM-index but is comparable in many ways and outperforms run length compressed FM-indexes for highly repetitive data [15].

All code except RLCSA[1] and the hybrid bitvector are part of the `sdsl-lite`[2] library. All optional structures not needed for count queries are excluded.

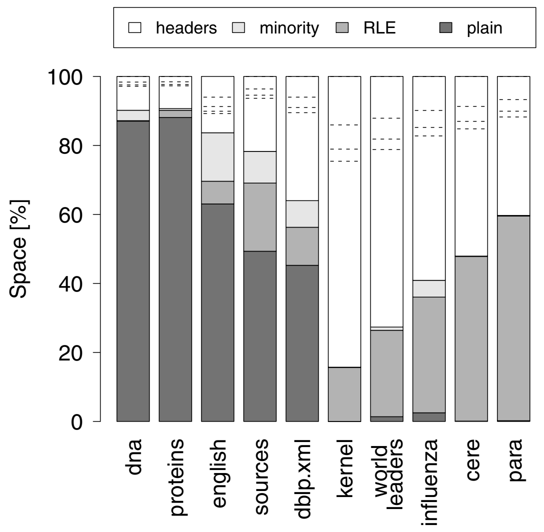
Figure 1 shows the size and query times for the indexes. For all data sets, FM-hybrid is among the smallest and among the fastest indexes. On the most repetitive data sets the run-length compressed indexes slightly outperform FM-hybrid, but they also perform very poorly on non-repetitive data. FM-RRR variants occasionally compress slightly better but often significantly worse, and are always much slower. FM-uncompressed is usually faster but offers little in the way of compression.

In Figure 2, we take a closer look at what is happening inside the hybrid bitvector. The graph on the right classifies the blocks according to how their body is encoded. One of the categories is an empty body, from which are further separated blocks belonging to a uniform superblock of all ones or all zeros. In the latter case, not even the block header is accessed (ever), which makes rank very fast. The graph on the left shows how much of the total size is taken by bodies of each type and how much is taken by the block and superblock headers.

The graphs show the dramatic difference between highly repetitive and non-repetitive data. For highly repetitive data, almost all blocks either have an empty body or the body is run-length encoded. The headers form a big part of the space usage and significantly better compression could be achieved by increasing the block size. The space usage of non-repetitive data is dominated by uncompressed blocks. There may be potential for improving the compression by replacing some of the uncompressed blocks with RRR compression or some other compression method.

[1]`http://www.cs.helsinki.fi/group/suds/rlcsa/`

[2]`https://github.com/simongog/sdsl-lite`



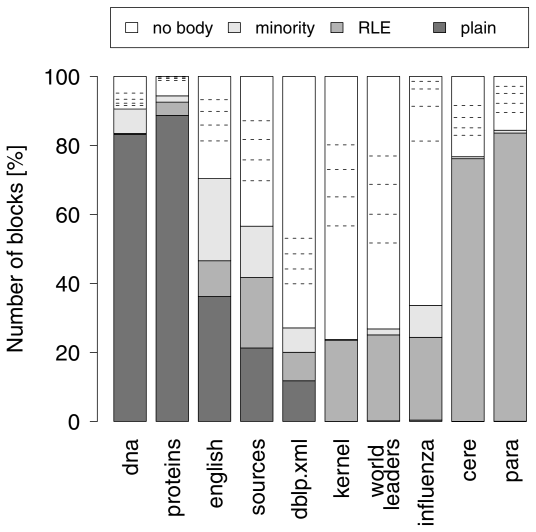


Figure 2: Left: distribution of space among different components of a hybrid bitvector used in a Huffman-shaped wavelet tree. The white block shows a share of headers assuming $b_s = 8$. The dotted lines show the reduction in header space when increasing b_s to to 16, 32 and 64. Right: the proportions of blocks encoded by different methods. Blocks consisting of at most two runs ("no body") are partitioned according to how many blocks lie within a uniform superblock. Each dotted line corresponds to a choice of $b_s \in \{8, 16, 32, 64\}$, starting with the bottom line, with the proportion of blocks in a uniform superblock above the dotted line.

5 Concluding Remarks

The hybrid bitvector described in this paper should be considered as a first demonstration of the concept of hybrid encoding. As the discussion in the preceding sections shows, the possibilities of hybrid encoding with different compression methods and different design choices are nearly endless. Different applications may require quite different types of hybrid bitvectors. However, the good all-around performance of our hybrid bitvector suggests that a well-designed hybrid bitvector might work well for many different purposes.

References

[1] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

[2] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.

[3] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Experiment. Algor.*, 13:1.12–1.31, 2009.

[4] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[5] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):Article 20, 2007.

[6] P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nat. Methods*, 6(11 Suppl):S6–S12, Nov. 2009.

[7] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software Pract. Exper.*, 2013. `doi:10.1002/spe.2198`.

[8] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. SIAM, 2003.

[9] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. SODA*, pages 636–645. SIAM, 2004.

[10] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.

[11] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. SPIRE*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011.

[12] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[13] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM*, volume 3537 of *LNCS*, pages 45–56. Springer, 2005.

[14] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE*, volume 4726 of *LNCS*, pages 229–241. Springer, 2007.

[15] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308, 2010.

[16] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[17] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[18] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, volume 7276 of *LNCS*, pages 295–306. Springer, 2012.

[19] G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. SEA*, volume 6630 of *LNCS*, pages 193–205. Springer, 2011.

[20] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, pages 60–70. SIAM, 2007.

[21] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4), 2007.

[22] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Proc. SEA*, volume 7933 of *LNCS*, pages 151–163. Springer, 2013.