SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Optimized Succinct Data Structures for Massive Data

Simon Gog[1]* and Matthias Petri[2]

[1]*Department of Computing and Information Systems, The University of Melbourne, VIC, 3010, Melbourne, Australia*
[2]*School of Computer Science and Information Technology, RMIT University, VIC, 3001, Melbourne, Australia*

SUMMARY

Succinct data structures provide the same functionality as their corresponding traditional data structure in compact space. We improve on functions $rank$ and $select$, which are the basic building blocks of FM-indexes and other succinct data structures. First, we present a cache-optimal, uncompressed bitvector representation which outperforms all existing approaches. Next, we improve — in both space and time — on a recent result by Navarro and Providel on compressed bitvectors. Last we show techniques to perform $rank$ and $select$ on 64-bit words which are up to three times faster than existing methods. In our experimental evaluation we first show how our improvements affect cache and runtime performance of both operations on data sets larger than commonly used in the evaluation of succinct data structures. Our experiments show that our improvements to these basic operations significantly improve the runtime performance and compression effectiveness of FM-Indexes on small and large data sets. To our knowledge, our improvements result in FM-indexes that are either smaller or faster than all current state of the art implementations. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: succinct data structures, binary sequences, FM-Index, algorithm engineering, massive data sets, rank, select, sse, hugepages

1. INTRODUCTION

In fields such as bioinformatics, information retrieval or data mining, massive volumes of information are processed on a regular basis. One of the biggest challenges in efficiently processing data of this size are in-memory storage capacity constraints: standard workstations cannot keep the complete data sets in-memory [14]. A potential remedy to this problem is the use of space-efficient — *succinct* — data structures. Succinct data structures take space close to the information theoretic lower bound needed to represent the underlying objects while providing the same functionality as their classical counterparts. Succinct data structures have successfully been used in the fields such as information retrieval and bioinformatics [4, 19].

Prominent examples of succinct data structures include compressed representations of suffix arrays and suffix trees. An uncompressed suffix tree requires the storage of multiple, say k, pointers per node; so each node requires $k \log n$ bits, and a suffix tree itself consists in the worst-case of $2n - 1$ nodes for a text of size n. In comparison, a succinct representation of a suffix tree can be represented in only $2n + o(n)$ bits. This translates to several orders of magnitude less space used in practice [10, 25]. Consequently, succinct data structures can be used to process texts an order of

*Correspondence to: E-mail: simon.gog@unimelb.edu.au

Contract/grant sponsor: The first author was supported by the Australian Research Council. The second author by NICTA.

magnitude larger than classical data structures. Another popular succinct data structure is the FM-index [7], which emulates a suffix array. To perform search, the suffix array requires $n \log n$ bits in addition to the original text T, while the FM-index uses space roughly equal to the size of the compressed representation of T. The FM-index efficiently supports the following operations over a compressed representation of a given text T: (1) $count$ the number of occurrences of a pattern P in T; (2) $locate$ all positions of P in T; (3) $extract\ T[i..j]$ from the index. All of these operations can be implemented as a combination of multiple basic operations on bitvectors which we will focus on in this paper [8].

In addition to memory constraints, which can be mitigated by succinct data structures, processing power has also become a bottleneck for computationally large tasks. For years programmers could rely on CPU manufacturers increasing processor speeds with every generation. Unfortunately, this is no longer the case, and the speed of processors has been stagnant for the last five years. New processor generations, in recent years, provide better performance through including multiple processing cores which, unfortunately, do not affect single threaded program execution. The instruction set supported by current processors has also become more versatile. Additional instruction sets such as SSE 4.2 can perform critical operations more efficiently. However, they require time and effort by the programmer to be effective [13, 30]. Broadword programming is another optimization on the word (or larger) level [17] which has already been successfully applied to the field of succinct data structures [31].

Another emerging problem is the cost of memory access. Each memory access performed by a program requires the operating system to translate the virtual address of a memory location to its physical location in RAM. All address mappings are stored in a process specific page table in RAM itself. Today, all CPUs additionally contain a fast address cache called the Translation Lookaside Buffer (TLB). The TLB is used to cache address translation results similar to the way the first level (L1) cache is used to store data from recently accessed memory locations. If the TLB does not contain the requested address mapping, the in-memory page table has to be queried. This is generally referred to as a TLB miss and similar to a L1/L2 cache miss, affects runtime performance. Accessing main memory at random locations results in frequent TLB misses as the amount of memory locations cached in the TLB is limited. Operating systems provide features such as hugepages to improve the runtime performance of in-memory data structures. Hugepages allow the increase of the default memory page size of 4 kB to up to 16 GB which can significantly decrease the cost of address translation as a larger area of memory can be "serviced" by the TLB. Succinct data structures such as FM-indexes exhibit random memory access patterns when performing operations such as $count$, yet to our knowledge, the effect of hugepages on the performance of succinct data structures has not yet been explored.

Unfortunately, in practice, the runtime of operations on succinct data structures tends to be slower than the original data structure they emulate. Succinct data structures should therefore only be used where memory constraints prohibit the use of traditional data structures. Twenty years ago Ian Munro conjectured that we no longer live in a 32-bit world [21]. Yet, many experimental studies involving succinct data structures are still based on small data sets, which contradicts the original motivation for their development. Furthermore, the cost of memory translation increases as the size of the in-memory data structure increases, and this effect cannot be clearly measured when performing experiments on small data sets.

In this paper we focus on improvements to basic operations on bitvectors used in many succinct data structures. We specifically focus on the performance on data sets much larger than in previous studies. We explore cache friendly layouts, new architecture dependent parameters and previously unexplored operating system features such as hugepages. Specifically, we show that our improvements to these basic operations translates to significant performance improvements of FM-type succinct text indexes. Our contributions can be summarized as follows:

- We propose a cache efficient, uncompressed rank enabled bitvector representation which, as data sets become larger, approaches the time it takes to $access$ an individual bit.
- We provide a practical select implementation based on Clark's proposal [2], and show it to have low space overhead, and fast query and construction time for many kinds of bitvectors.

DOI: 10.1002/spe

- We present optimizations on compressed bitvectors and show that we can improve the recent results of Navarro and Providel [24].

We further provide an extensive empirical evaluation by building on the experimental studies of Ferragina et al. [6], Vigna [31] and González et al. [11]. We explore the following aspects.

- We show that previously unexplored, machine dependent features can be used to increase the performance of succinct data structures dramatically.
- To the best of our knowledge we are the first to explore the behavior of these succinct data structures on massive data sets (64 GB compared to commonly used 200 MB)
- We then show how the speed-up of basic bit operations propagates through the different levels of succinct data structures: from binary $rank$ and $select$ over bitvectors to FM-indexes.

The paper is structured as follows: We first give an overview of previous work. Next we introduce our newly engineered uncompressed bitvector representations for $rank$ in Section 3 and $select$ in Section 4. Further, we discuss practical improvements to the compressed bitvector representation of Navarro and Providel in Section 5. Last we provide an extensive empirical evaluation of our improvements in Section 6.

2. DEFINITION AND PREVIOUS WORK

In this paper we consider the space-efficient implementation of the following operations on a vector $B[0..n-1]$ of length n over an alphabet Σ of size σ:

$access(B, i)$:	Return the i-th element of sequence B, also denoted as $B[i]$.
$rank(B, i, c)$:	Return the number of occurrences of symbol c in the prefix $B[0..i-1]$.
$select(B, i, c)$:	Return the position of the i-th occurrence of symbol c in B.

We are especially interested in the efficient implementation over bitvectors ($\Sigma = \{0, 1\}$,$\sigma = 2$), since the case of larger alphabets can be reduced to $\log \sigma$ operations on bitvectors using a wavelet tree [12]. A wavelet tree recursively divides the alphabet σ to create a binary tree whose notes correspond to individual symbols in the alphabet. General $rank(B, i, c)$ queries over σ can be answered in $\mathcal{O}(\log \sigma)$ time by traversing the tree to the leaf level. Changing the underlying bitvectors or the shape of the tree can lead to a variety of different time-space trade-offs [6]. For example, using Huffman codes to encode each symbol in B results a Huffman-shaped wavelet tree which tends to be more compressible. For an extensive overview on wavelet trees see [22].

Let m be the number of set bits in B. In the case of an uncompressed bitvector, $access$ takes constant time. Jacobson [2] and Clark [15] have showen that it is possible to answer $rank$ and $select$ also in constant time when using $o(n)$ bits of additinal space. It is also possible to replace B by an entropy compressed representation without increasing the time bounds [28]. The latter approach is used in practice for *sparse* ($m \ll n$) and the former for *dense* bitvectors. Implementations for the dense case were presented by González et al. [11] and Vigna [31]. For the sparse case Claude and Navarro [3], and recently Navarro and Providel [24] presented implementations of [27, 28]. Okanohara and Sadakane [26] presented an implementation for very sparse bitvectors following the idea of [5]. Most implementations reduce $rank(B, i, c)$ to performing population count ($popcnt$), that is counting the number of set bits, on a specific computer word in B. The $select(B, i, c)$ operation is generally reduced to determining the position of the j-th bit in a single computer word. We later refer to this operation as $select_{64}$ because bitvectors in our implementations are represented as a sequence of 64-bit integers.

Note, that most of these studies did not explore the effects on massive data. González et al. perform experiments on bitvectors of sizes up to 128 MB [11], Navarro et al. use bitvectors of size 32 MB and texts of size 50 MB [24], Okanohara and Sadakane [26] instance up to 120 MB.

DOI: 10.1002/spe

Vigna [31] was the first who used data in the gigabyte range and provided a 64-bit implementation in an easy-to-use library[†].

The most common use for succinct data structures is in text indexing. Traditionally, suffix arrays (SA) are used to perform text indexing [20]. A suffix array $\mathsf{SA}[0 \dots n-1]$ stores, in $n \log n$ bits, all suffix positions of a text T in lexicographical order. That is, the suffix position stored at $\mathsf{SA}[i]$ is lexicographically smaller than the suffix starting at position $\mathsf{SA}[i+1]$: $T[\mathsf{SA}[i] \dots n-1] < T[\mathsf{SA}[i+1] \dots n-1]$. All occurrences of a pattern in the text can then be found efficiently in a continuous range $\mathsf{SA}[i \dots j]$. Succinct text indexes try to emulate the functionality of suffix arrays and suffix trees in succinct space [23]. One popular succinct text index is the FM-index initially proposed by Ferragina and Manzini [7]. The FM-index exploits a duality between the suffix array and the Burrows-Wheeler Transform (BWT) where $T^{bwt}[i] = T[\mathsf{SA}[i]-1]$. The BWT permutes T by lexicographically sorting all rotations of T to produce T^{bwt}. The transform was initially used in data compression systems as T^{bwt} tends to be more compressible than T while the original text can be recovered from T^{bwt} in linear time without required any additional information [1]. FM-indexes can be implemented efficiently using $rank$ and $select$ operations over T^{bwt} using a wavelet tree [8]. An FM-index supports the following operations:

$count(P)$:	Return the number of times pattern P occurs in $T[0..n]$.
$locate(P)$:	Return all positions of pattern P in $T[0..n]$.
$extract(i,j)$:	Extract $T[i..j]$ from the index.

The key component of all operations supported by the FM-index is *backwards search*. Backwards search allows, for a pattern P of length $|P|$, determination of the range $\mathsf{SA}[\ell..r]$, where all suffixes are prefixed by P, using $2|P| \log \sigma$ basic $rank$ operations using a wavelet tree. Over time, several other succinct text indexes based on the initial concept of the FM-index have been proposed. For example, Mäkinen and Navarro combine a run-length encoded wavelet tree of T^{bwt} and two additional bitvectors [18]. Unlike a normal FM-index, run-length based FM-indexes (RLFM) require additional $select$ operations to perform backward search.

Ferragina et al. perform an extensive evaluation of compressed text indexes by providing multiple baseline implementations and a widely used test corpus [6]: The *Pizza&Chili* Corpus available at `http://pizzachili.dcc.uchile.cl/texts.html`. The corpus consists of files up to 2 GB in size from different domains and has been used by many researches as a reference collection in experiments [3, 16, 24].

3. A CACHE FRIENDLY RANK IMPLEMENTATION FOR UNCOMPRESSED BITVECTORS

The first sub-linear-space data structures supporting constant time $rank$ operations store two levels of precomputed rank values: *blocks* R_b and *superblocks* R_s [15, 21]. In theory, each superblock $R_s[i]$ stores pre-computed $rank$ values at $s = \log^2 n$ intervals at a total cost of $n/\log n$ bits. Each block $R_b[i]$ stores precomputed $rank$ values relative to the preceding superblock in $b = \log n$ intervals at a total cost of $n \log\log n/\log n$ bits. This results in a total cost of $n/\log n + n\log\log n/\log n \in o(n)$ bits.

In practice, $rank(B,i,1)$ is answered as follows. First, the correct superblock $R_s[i/s]$ is retrieved. Next, the correct block $R_b[i/b]$ is calculated. Finally, $rank(B,i,1) = R_s[i/s] + R_b[i/b] + popcnt(bv[v])$, where $bv[v]$ is the machine word containing position i masked up to position i, and $popcnt$ a method which counts the ones in the word. Note that each choice of the parameters s and b implies a certain level of overhead. For example, 38% overhead is achieved for $b = 32$ and $s = b \log n$.

In addition to fast, basic bit-operations, minimizing cache and TLB misses is the key to fast succinct data structures [11, 31]. Using two block levels may result in 3 cache misses to answer

[†]Available at: `http/sux.dsi.unimi.it`.

DOI: 10.1002/spe

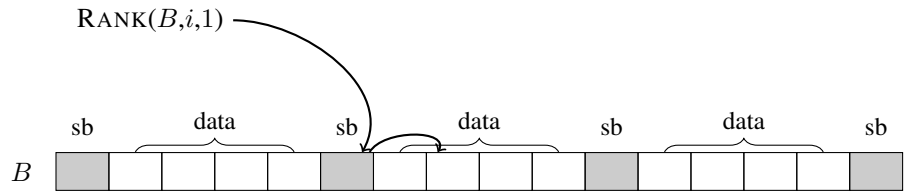

Figure 1. Interleaved bitvector (B) showing superblocks (sb) and data interleaved for optimal cache performance. A $rank$ operation first jumps to the interleaved superblock and then iteratively processes the following data block.

a rank query: one access to R_s and R_b each and a third access to the bitvector to perform $popcnt$ within the target block. To reduce cache and TLB misses, Vigna proposed to *interleave* R_s and R_b [31]. So each superblock $R_s[i]$ which contains absolute rank samples is followed in memory by the corresponding blocks $R_b[j..k]$ containing the relative rank samples. The size of the blocks are chosen as follows: 64 bits are used to store each superblock value. The second 64 bits are used to store seven 9-bit block counts. Overall, $rank$ queries can be answered in constant time for a superblock size of $s = (7 + 1) \cdot 2^9 = 512$. This reduces the number of cache and TLB misses to two: one to access the precalculated counts and a second to perform the population counts within the bitvector. The total space overhead of this approach is $128/512 = 25\%$. We refer to this approach as RANK-V. Note that it is called RANK9 in [31].

Extending Vigna's approach, we propose interleaving the pre-computed $rank$ values and the bitvector data as shown in Figure 1. Furthermore, we only store one level of pre-computed values similar to one of the methods proposed in [11]. For large enough bitvectors and fast $popcnt$ methods we conjecture that this will increase run time performance as we perform only one cache and TLB miss. For each block b we therefore only store a 64-bit cumulative rank count. For a block size of 256 bits, the space overhead of our method is 25%, the same as the solution proposed by Vigna. We call our our 25% overhead implementation RANK-1L.

More space-efficient versions can be achieved by choosing a larger block size. For example a block size of 1024 results in 6.25% extra space. The same space can also be achieved by a variation of Vigna's approach, which we call RANK-V5. Set the superblock size to $s = 2048$ and the block size to $b = 6 \cdot 64 = 384$ bits [31]. RANK-V5 stores the superblock values in a 64-bit word and the 5 positive relative 11-bit counts in a second 64-bit word. A rank query then requires two memory accesses plus at most 6 $popcnt$ operations.

4. A FAST SELECT IMPLEMENTATION FOR UNCOMPRESSED BITVECTORS

Using the block/superblock data structure described in Section 3, a $select(B, i, 1)$ operation can be solved by performing $\mathcal{O}(\log n)$ $rank$ operations over a bit vector to determine the position of the i-th one bit. González et al. [11] show that using binary search over the prestored superblock counts R_s followed by sequential search within the determined superblock is several time faster. In their one level $rank$ structure containing only superblocks, González et al. sequentially search within the superblock by performing byte-wise $popcnt$ operations followed by a bitwise search.

We implement this approach over RANK-1L and call it SEL-BS. However, performing binary search for large inputs is inefficient as the array positions of the first binary search steps lie far apart and consequentially cause cache and TLB misses. We provide a second implementation called SEL-BSH, which uses an auxiliary array H. The array contains 1024 rank samples of the first 10 steps of each possible binary search in heap-order. SEL-BSH uses H to improve memory access locality of the first 10 steps of the binary search to reduce TLB and cache misses.

Clark presents a sublinear space ($\frac{3n}{|\log\log n|} + \mathcal{O}(\sqrt{n}\log n \log\log n)$ bits), constant time 3-level data structure for $select$ in his PhD thesis [2, Section 2.2.2, pp. 30-32]. González et al.'s [11]

DOI: 10.1002/spe

verbatim implementation of Clark's structure requires 60% percent space overhead in addition to the original bitvector and was slower than SEL-BS for inputs smaller than 32 MB.

We now present an implementation of a simplified version of Clark's proposal, called SEL-C. Let $m \leq n$ be the number of ones in the bitvector B. We divide B in *superblocks* by storing the position of every 4096-th 1-bit. Each position is stored explicitly at a cost of $\lceil \log n \rceil \leq 64$ bits. In total we use $\lceil \frac{n}{4096} \rceil \cdot \log n$ bits of extra space. When using 64 bit words to store each position, the additional space required is $\lceil \frac{64 \cdot n}{4096} \rceil$ bits or 1.6%. Using this structure we can answer $select$ queries for every 4096-th 1-bit directly. Let $x = select(4096i + 1)$ and $y = select(4096(i + 1) + 1)$ be the border position of the i-th superblock. A superblock is called *long*, if its size $r = y - x$ is larger or equal $\log^4 n$. As a result, storing each of the 4096 positions explicitly requires only $4096/\log^3 n$ bits per position which translates to only $\leq 10\%$ overhead per bit in a 1 GB bitvector. If $r < \log^4 n$, we call the superblock *short,* and it is further subdivided by storing the position of every 64-th 1-bit relative to the left border of the superblock. This requires at most $\log r \leq \log(\log^4 n) = 4 \log \log n$ bits per entry. Hence an upper bound for the space overhead is $\frac{4096}{64} \cdot \log r = 64 \cdot \log r$, which in the worst case ($r = 4096$) results in an overhead of $\frac{64 \cdot \log r}{4096} = \frac{64 \cdot 12}{4096} = 18.75\%$. For the important case of *dense bitvectors*, the typical space overhead is much smaller than the worst case, since $r \approx 2 \cdot 4096$ which translates into $\frac{64 \cdot \log(8196)}{8196} = 10.2\%$ overhead per input bit.

Vigna also proposed two $select(B, i, 1)$ implementations called SELECT9 and SIMPLE which we refer to as SEL-V9 and SEL-VS [31]. SEL-V9 builds select capabilities on top of RANK-V. In addition to the $rank$ counts, a two level inventory is stored at a total cost of 57.5% space overhead. SEL-VS similar to SEL-V9, uses a multi-level inventory to perform select but does not require RANK-V during query time which results in less space-overhead. Similar to the select structure of Clark [2], a three level structure is used in SEL-VS. However, SEL-VS is algorithmically engineered to use only 64-bit and 16-bit words instead of using integers of size $\log r$ which are more expensive to access and store. For very large superblocks, SEL-VS further stores the absolute positions of each set bit using a 64-bit word, whereas Clark's structure stores the position in $\log r$ bits relative to the beginning of the superblock. For smaller superblocks, the positions are not stored explicitly. Instead, two inventories are used to find a region in the original bitvector which is then scanned sequentially to find the correct i-th bit position.

5. ENGINEERING A BETTER H_0-COMPRESSED BITVECTOR

Sparse bitvectors, in which the number of ones, m, is significantly smaller than $n/2$, can be stored more succinctly using a bitvector representation such as that proposed by Pagh [27]. The representation requires $\lceil \log \binom{n}{m} \rceil + \mathcal{O}(\log \log n) + o(n)$ bits in theory, and supports $access$ and $rank$ in constant time. Note that the first term above is bounded by the zeroth order entropy H_0 of the bitvector using Stirling's formula.

We will briefly revisit the data structure to explain our practical optimizations. To achieve the described compression, the original bitvector B is divided into $n' = \lceil n/K \rceil$ blocks of fixed length K. The information of each block b_i is split into two parts: First the number κ_i representing the number of one bits in the block. There are only $\binom{K}{\kappa_i}$ possible bit-permutations of b_i containing κ_i ones. Therefore, storing a second number $\lambda_i \in [0, \binom{K}{\kappa_i} - 1]$ is enough to uniquely encode and decode block b_i. Each κ_i is stored in an array $\mathsf{C}[0 \dots \frac{n}{K}]$ of $\lceil \log(K + 1) \rceil$-bit integers. Compression is achieved by representing each λ_i with only $\lceil \log(\binom{K}{\kappa_i} + 1) \rceil$ bits and storing all λ_i consecutively in a bitvector O. To efficiently answer $access$ and $rank$ queries, samples, with a sample rate t, are stored in an interleaved array S. For each block b_{jt} the element $\mathsf{S}[j]$ contains the starting position of λ_{jt} and the rank to the start of the block $rank(B, jtK, 1)$.

Performing Operations $access(B, i)$ or $rank(B, i, 1)$ can be answered in time $\mathcal{O}(t)$ as follows. First we determine the block index $i' = \lfloor i/K \rfloor$ of bit i. Second, we calculate the intermediate block $\tilde{i} = \lfloor \frac{i'}{t} \rfloor t$ prior to i' which contains a pointer into O. Third, the sum Δ of the binary lengths of each

DOI: 10.1002/spe

λ_j ($\tilde{i} \leq j \leq i'-1$) is calculated by sequentially scanning C, adding $\lceil \log(\binom{K}{\kappa_j} + 1) \rceil$ for each block b_j. Finally, $b_{i'}$, the block containing position i, can be reconstructed by accessing $\kappa_{i'}$ and $\lambda_{i'}$ directly as they are encoded at position $\mathsf{S}[\tilde{i}]+\Delta$ with $\lceil \log(\binom{K}{\kappa_{i'}} + 1) \rceil$ bits in O. Having recovered $b_{i'}$ from O and C, we can answer $access(B, i)$ by returning the bit at index $i \bmod K$. Operation $rank(B, i, 1)$ can be answered by adding $\mathsf{S}[\tilde{i}+1]$, the sum of values in $\mathsf{C}[t\tilde{i}..i'-1]$, and $rank(b_{i'}, i \bmod K)$.

In practice $select$ is performed in $\mathcal{O}(\log n)$ time by first performing binary search over the rank samples in S and then sequentially scanning the blocks between the target interval.

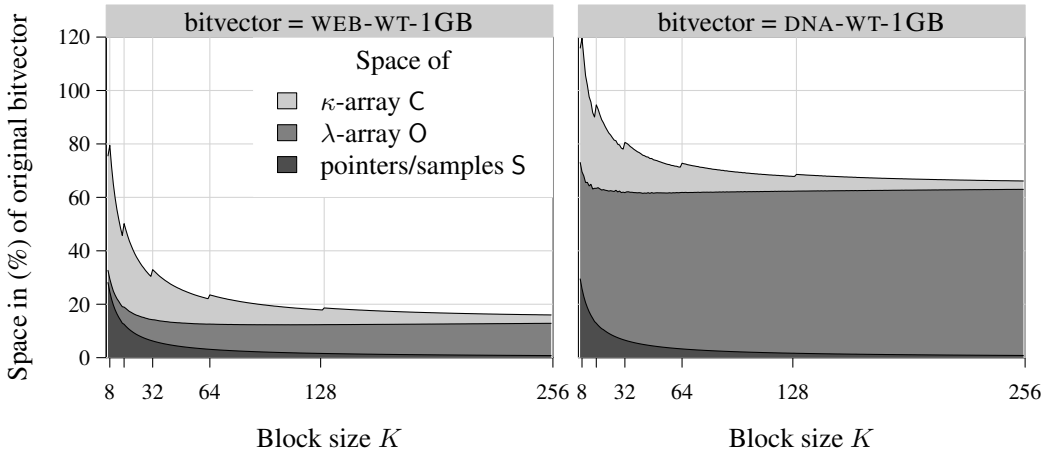

Figure 2. Space consumption of the three different parts of the H_0-compressed bitvector as a function of block size K for two different bitvectors. The sample rate t was set to 32. The original bitvectors of size 1 GB were extracted from wavelet trees of WEB and DNA text (see Section 6.1).

Space Consumption The total space of O is bounded by $nH_0 + \frac{n}{K}$ bits (see [27]). Bitvector C is bounded by $n\frac{\log K}{K}$ bits, and bitvector S by $2n\frac{\log n}{tK}$ bits. If the bitvector is compressible – that is $H_0 \ll 1$ – then the size of C is dominant when K is small. Figure 2 shows the space usage of the individual components (C,O,S) for two, "real world" bitvectors WEB-WT-1GB and DNA-WT-1GB; see to Section 6.1 for a detailed description of their origin. Using Claude et al.'s [3] implementation, which uses blocks of size $K = 15$ and decodes $b_{i'}$ from $(\kappa_{i'}, \lambda_{i'})$ via a lookup table, the sizes are $|\mathsf{C}|$ =275 MB and $|\mathsf{O}|$ =64 MB. Note that lookup-tables for larger K are not practical. Navarro and Providel [24] recently propose decoding and encoding of λ_i without a lookup table by encoding and decoding blocks in $\mathcal{O}(K)$ time on the fly. During the encoding process, λ_i is computed from a block b_i with κ_i set bits as follows: Initially $\lambda_i = 0$. First, the least significant bit in b_i is considered. There are $\binom{K-1}{\kappa_i}$ blocks ending on zero and $\binom{K-1}{\kappa_i-1}$ ending on one. If the last bit is one, λ_i is increased by $\binom{K-1}{\kappa}$ (that is the number of blocks of ending with zero); otherwise λ_i is not changed. In the next step, K is decreased by one, and κ_i is decreased by one if the last bit was set. b_i is shifted to the right and we reevaluate the least significant bit. The process ends when $\kappa_i = 0$.

A block b_i can be recovered from λ_i and κ_i as follows: If $\lambda_i \geq \binom{K-1}{\kappa_i}$, the least significant bit in b was set. In this case λ_i is decreased by $\binom{K-1}{\kappa_i}$ and κ_i by one. Otherwise the least significant bit was a zero. In the next step, K is decreased by one and repeat the decoding process until $\kappa_i = 0$.

On-the-fly decoding requires only $\mathcal{O}(K)$ simple arithmetic operations (subtraction, shifts and comparisons) and a lookup table of size K^2 to calculate the binomial coefficients. Navarro and Providel [24] use a block size of $K = 63$, which reduces the size of C to 98 MB. This is still of similar size as C (97 MB) for WEB-WT-1GB. It was reported that the use of $K > 63$ results in runtimes "orders of magnitudes slower" than for smaller K.

We try to improve on this result by applying the following optimizations:

DOI: 10.1002/spe

1. The $access$ operation can immediately return the result if block b_i contains only zeros or ones (called *uniform*), without scanning O and accessing S. The $rank$ operation can immediately return its result if the $(\tilde{i}+1)$-th and $\tilde{i}$-th rank sample in S differ by 0 or tK, saving the scanning of C. At first glance, this optimization seems to be trivial, however in text indexing the bitvectors of wavelet trees often contain long runs of zero and ones and we therefore conjecture that this optimization will be effective in practice.
2. If a block $b+i$ has to be decoded, then we can apply an optimization if the block contains only few ones: We first check if $\kappa_i \leq \frac{K}{\log K}$. In this case it is faster to determine the κ_i positions of ones in b_i by κ_i binary search over the columns in Pascal's triangle.

6. EXPERIMENTAL STUDY

6.1. Experimental Setup

Hardware Our main experimental machine is a server equipped with $2\times$ Intel Xeon E5640 processors each with a 12 MB L3 cache. The total memory is 144 GB of DDR3 DRAM, 72 GB directly attached to each socket. This implies that the memory architecture is non-uniform (NUMA) as accessing a different NUMA domain (the memory attached to a different socket) is more expensive. Each processor uses 32 kB L1 data cache and 256 kB L2 cache per core. The L1 cache line size is 64 bytes. The processor additionally supports memory page sizes of 4 kB, 2 MB and 1 GB. All our experiments are run on a single thread. Each CPU further includes a two level translation lookaside buffer (TLB) with 64 times 4 kB-page entries in the first level and 512 in the second level. The TLB has 4 dedicated entries for 1 GB-pages in its first level.

We also used a second machine to sanity check our results on small test instances. A MacBook Pro equipped with a Intel Core i5-2435M Processor with 3 MB L3 cache, 4 GB of DDR3 DRAM. The processor uses 64 kB L1 data cache and 256 kB L2 cache per core.

Software Ubuntu Linux version 12.04 served as operation system on the server, while Mac OS X version 10.7.3 was used on the MacBook. We used the `g++` compiler version 4.6.3 on the server and version 4.3.6 on the MacBook. The compiler flags `-O3 -DNDEBUG -funroll-loops` were used to compile all programs. We selectivly enabled SSE 4.2 support using the `-msse4.2` flag.

All our implementations are included in the open source C++ template library sdsl which is available at `http://github.com/simongog/sdsl`. The library includes a test framework which was used to check the correctness of our implementations. The library also provides methods to measure the space usage of data structures and a provides support for timings based on the `getrusage` and `gettimeofday` functions. We state the elapsed *real* time in all results.

The *papi* library version 4.4.0, which is capable of reading performance counters of CPUs, was used to measure cache and TLB misses. It is available under `http://icl.cs.utk.edu/papi/`. Note that we deliberatively choose to not use *cachegrind*, a cache measurement tool provided in the *valgrind* debugging and profiling tool suite. Cachegrind performs simulations to determine the number of cache misses caused by a program. However, this simulation does not elucidate cache misses and TLB misses caused when accessing the page table, which is also stored in memory. As the size of the page table grows with the size of the data, cache misses and TLB misses resulting from accesses to the page table become more frequent and thus more relevant to the overall performance of a data structure.

Environmental Features One of our main goals is to study how much an implementation can be improved varying the feature set of the operating system and hardware employed. We have selected the following three features:

DOI: 10.1002/spe

TBL Programs marked with this feature use a lookup table to perform $popcnt$ on bytes and process the final byte of a $select_{64}$ operation bit by bit. Compiling with flag `-DPOPCOUNT_TL` activates the feature.

BLT A program marked with BLT uses the CPU built-in $popcnt$ function and functions to determine the leading and trailing zeros. It also activates the use of the new $select_{64}$ implementation presented in Section 6.2. The feature can be used by adding the compile flag `-msse4.2` for `gcc` versions ≥ 4.3.

HP Programs marked with this feature use 1 GB memory pages instead of the standard 4 kB pages. This feature is supported by, among other operating systems, Linux kernels in the 2.6 series or newer. However, the usage of this feature requires preparation. Hugepages must be reserved at boot time and can then be used by memory mapping their physical memory area. For this study we have implemented a basic *memory management system* in the sdsl library to be able to map the heap allocated part of our data structures to hugepages. The library keeps track of all allocated memory and maps all structures when `sdsl::mm::map_hp()` is called. After the execution of an experiment the structure can be copied back to 4 kB page memory region by a call of `sdsl::mm::unmap_hp()`. The use of 1 GB pages reduces the number of TLB misses for programs which perform non-local memory accesses.

Data Sets As in previous studies on $rank$ and $select$ [11, 31, 24], we created bitvectors of varying sizes and densities to evaluate our data structures. The instance sizes range from 1 MB to 64 GB, quadrupling in size each time. For each size we generate 3 vectors with density $d = 5\%$, $d = 20\%$, and $d = 50\%$ by setting each bit with probability d.

For the evaluation of the FM-indexes we used the following data sets:

- The 200 MB instances (DNA,DBLP.XML,PROTEINS,SOURCES, ENGLISH) of the *Pizza&Chili* website, which provides texts from different application areas and is the standard benchmark in text indexing. The 200 MB test instances are the largest files which are available for all categories on the website. We also created bitvectors denoted F-WT-X, where each has size X and was produced by taking a prefix of the instance file F, calculating the Huffman-shaped wavelet tree WT of the Burrows-Wheeler-Transform of this prefix and concatenating all the bitvectors of WT.
- To go beyond the 200 MB limit, we created a 64 GB file from the 2009 CLUEWEB web crawl available at `http://lemurproject.org/clueweb09.php/`. The first 64 files in the directory `ClueWeb09/disk1/ClueWeb09_English_1/enwp00/` were concatenated and null bytes in the text were replaced by 0xFF-bytes. This file is denoted WEB-64GB. Prefixes of size X are denoted by WEB-X. We again created bitvectors denoted by WEB-WT-X. Each has size X and was produced by taking a prefix of WEB-64GB, calculating the Huffman-shaped wavelet tree WT of the Burrows-Wheeler-Transform of this prefix and concatenating all the bitvectors of WT.
- Our second large text is a 3.6 GB genomic sequence file called DNA. It was created by concatenating the "Soft-masked" assembly sequence of the human genome (hg19/GRCH37) and the Dec. 2008 assembly of the cat genome (catChrV17e) in fasta format. We removed all comment/section separators and replaced them with a separator token to fix the alphabet size at $\sigma = 8$. We created bitvectors DNA-WT-X with the same process described for WEB-64GB.

Basic Data Structure Implementations In sdsl we distinguish between bitvectors and *support data structures* which add $rank$ or $select$ functionality. The uncompressed bitvector bv (class `bit_vector`) is represented as a sequence of 64-bit integers and only provides the $access$ operation. Operations $rank$ and $select$ can be answered by adding support data structures which are compatible with the bitvector class. We can therefore evaluate different time and space trade-offs for $rank$ and $select$ over a single bitvector. For example, class `bit_vector` can be supported by the 25% overhead class `rank_support_v<>` or by the 6.25% overhead class `rank_support_v5<>`, and if required

Prepared using speauth.cls
DOI: 10.1002/spe

Table I. Overview of *rank* and *select* implementations used in the experiments.

Name	Described in	Corresponding sdsl	
		support class	bitvector class
RANK-V	Sec. 3,[31]	`rank_support_v<>`	`bit_vector`
RANK-V5	Sec. 3	`rank_support_v5<>`	`bit_vector`
RANK-1L	Sec. 3	`rank_support_interleaved<1,256>`	`bit_vector_interleaved<256>`
RANK-R^3K	Sec. 5,[24]	`rrr_rank_support<1,`K`>`	`rrr_vector<`K`>`
RANK-SD	[26]	`sd_rank_support<>`	`sd_vector<>`
SEL-C	Sec. 4	`select_support_mcl<>`	`bit_vector`
SEL-BSH	Sec. 4	`select_support_interleaved<1,256>`	`bit_vector_interleaved<256>`
SEL-BS	Sec. 4	see SEL-BSH (compile option `-DNOSELCACHE`)	
SEL-R^3K	Sec. 5,[24]	`rrr_select_support<1,`K`>`	`rrr_vector<`K`>`
SEL-SD	[26]	`sd_select_support<>`	`sd_vector<>`
SEL-VS	[31]	—	
SEL-V9	[31]	—	

select queries can be supported in constant time by adding an object of class `select_support_mcl`. Table I summarized all our *rank* and *select* implementations used in the following experimental study, and references to where they were described in more detail in this paper or in previous work.

Random Queries In the first part of our experimental evaluation we perform random queries (*rank* or *select*) over different bitvectors. As many of these basic operations only take several nanoseconds, generating a random query position (using for example `rand()`) can affect the outcome of the experiments. We therefore took the approach of generating 2^{20} random numbers before we perform each measurement. Each random number represents one query position in the bitvector. We then sequentially perform multiple passes over the array holding the random numbers to perform the random *rank* or *select* operations on the bitvectors. The time shown for individual queries is the mean query time over 10^7 random queries. We sequentially cycle the random number array to minimize the effect of accessing the random numbers on the TLB/L1 miss rates in our experiments while not affecting the time measurements. To ensure this assumption is true we conducted experiments using a pseudo random number generator and observed only minor improvements in TLB performance. In our uncompressed *rank* experiments we further include the TLB and L1 Cache performance of a single random *access* operation on a bitvector as a baseline at the same 10^7 positions.

6.2. Rank and Select on 64*-bit words*

Answering operations $rank(B, i, c)$ and $select(B, i, c)$ requires computing $popcnt$ and $select_{64}$ on a 64-bit word. The first step to an optimal *rank* and *select* implementation is the optimization of these basic operations. In 2005, González et al. [11] found that computing $popcnt$ using multiple 8-bit probs into a loookup table resulted in the best performance. We refer to this apporach as TBL. In 2008, Vigna used a broadword computing technique discussed by Knuth [17, p. 143]. We call this approach BW. Recently, researchers started using SSE 4.2 instructions to perform population count more efficiently [13, 30]. At the end of 2008 both INTEL and AMD released processors with efficient built-in CPU instructions for $popcnt$ on 64-bit words (down to 1 CPU cycle; see, for example, [9]).

Table II shows a comparison of the described methods. In the experiment we performed 10^7 operations on random values for each method. As described in Section 6.1, we cycle through our array of random numbers to avoid generating random numbers on-the-fly. Note that the table lookup method TBL which was considered the fastest in 2005 now is roughly five times slower for $popcnt$ than using the builtin CPU instruction. The broadward technique is rougly 3 times slower. This is by no means an exhaustive comparison of all available $popcnt$ methods. For a more in-depth comparison of different population count methods see [13, 30].

DOI: 10.1002/spe

Table II. Time in nanoseconds for $popcnt$ and $select_{64}$ implementations.

Operation	Average time (ns)		
	TBL	BW	BLT
$popcnt$	5.12	2.80	1.04
$select_{64}$	19.22	11.60	6.40

Unfortunately, the development of efficient $select$ operations on 64-bit integers ($select_{64}$) has not been as rapid as for $popcnt$. There are no direct CPU instructions available to return the position of the i-th set bit. Method TBL solves the problem in two steps: first, the byte which contains the i-th set bit is determined by performing sequential byte wise popcounts using a version of TBL. The final byte is scanned bit-by-bit to retrieve the position of the i-th bit. Vigna proposed a broadword computing method [31][‡] which we refer to as BW. In Lising 1 we introduce a faster variant of Vigna's method (BW) which uses two small additional lookup tables and a builtin CPU instruction.

```
 1 uint32_t select64(uint64_t x, uint32_t i) {
 2     uint64_t s = x, b, j;
 3     s = s-((s>>1) & 0x5555555555555555ULL);
 4     s = (s & 0x3333333333333333ULL)
 5         + ((s >> 2) & 0x3333333333333333ULL);
 6     s = (s + (s >> 4)) & 0x0F0F0F0F0F0F0F0FULL;
 7     s = s*0x0101010101010101ULL;
 8     b = (s+PsOverflow[i]) & 0x8080808080808080ULL;
 9     int bytenr = __builtin_ctzll(b) >> 3;
10     s <<= 8;
11     j -= ((uint8_t*)&s)[bytenr];
12     pos = (bytenr << 3) + Select256[((j-1) << 8)
13             + ((x>>(bytenr<<3))&0xFFULL)];
14     return pos;
15 }
```

Listing 1: Fast, branchless $select_{64}$ method using two lookup tables. The difference to Algorithm 2 in [31] is highlighted.

Given a 64-bit word x and a number i, we determine the position of the i-th set bit in x as follows. In the first step the byte b_i in x which contains the i-th set bit is determined. In the second step we use a lookup table to select the j-th set bit in b_i, which corresponds to the i-th set bit in word x.

In detail, we first use the divide and conquer approach described by Knuth which calculates the number of ones in each byte in x (lines 2-6). Line 7 calculates the cumulative sums of the byte counts using one multiplication. Next we use these cumulative sums to determine b_i. This can be done by adding a mask (`PsOverflow[`i`]`) depending on i to the cumulative sums (line 8). After the addition the most significant bit (MSB) of each byte is set, if its corresponding sum was larger than i (line 8). The first byte with a sum larger or equal to i contains the i-th bit in x. We use a second mask to set the remaining 7 bits in each byte to zero. Now the position of b_i corresponds to the number of trailing zeros divided by eight (line $9-10$). We use a CPU instruction (`builtin_ctzll`) to select the leftmost non-overflowed byte b_i. In the second step, we determine the rank j of the i-th bit in b_i by subtracting the number of set bits in x occurring before b_x (line 11). Last we use a lookup table (`Select256`) to select the j-th bit in b_i and return the position (pos) of j in the word x.

A performance comparisons of different $select_{64}$ methods on 64-bits is shown in Table II. Using incremental table lookups to determine the correct byte within the 64-bit integer and using bit-wise

[‡]available at `http://sux.dsi.unimi.it/select.php`

DOI: 10.1002/spe

processing of the target byte (TBL) is roughly 3 times slower than the fastest method BLT. The original broadword (BW) method is roughly 2 times slower than BLT.

6.3. Rank on Uncompressed Bitvectors

We now consider the two most efficient *popcnt* implementations (BW, BLT) and use them inside the fastest known implementation of $rank(B, i, c)$, RANK-V, and our new, fully interleaved representation RANK-1L. We also vary the memory page size since address translation can also affect the performance of the data structure. We use either the standard 4 kB-pages (no HP) or 1 GB-pages (HP).

The results of our random query experiment is depicted in Figure 3. Note that the performance is not affected by the density of the bitvector, since accessing the cumulative counts and performing *popcnt* does not depend on the underlying data. We further included the cost of performing a single $access(B, i)$ operation as a baseline and as a practical lower bound, since a *rank* operation cannot be faster than reading a single bit. Figure 3 shows that RANK-V is better than RANK-1L for

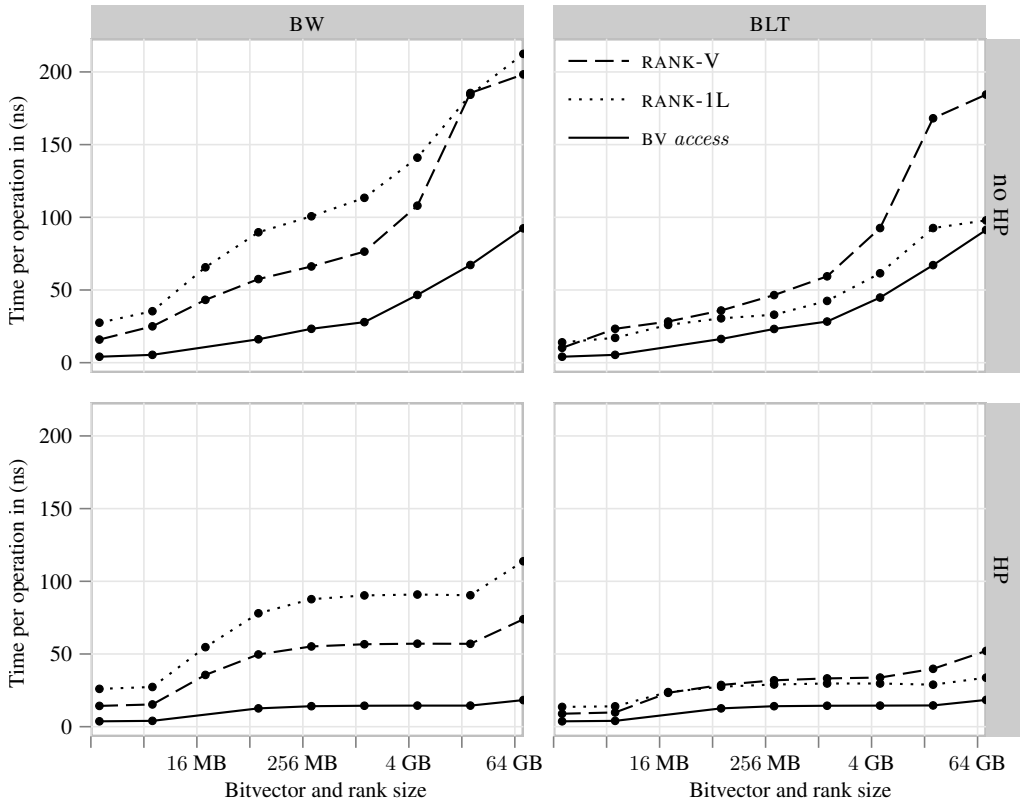

Figure 3. Time for single random *rank* operations on uncompressed bitvectors. The top row shows performance with standard 4 kB pages, the bottom row shows performance with 1 GB pages. The left column shows performance using the broadword method, and the right column with the BLT method.

test instances of larger size if we use *popcnt* method BW, and RANK-1L outperforms RANK-V if we use *popcnt* BLT. It also shows that the runtime of the *access* and *rank* operations significantly increases with the size of the data structure if the standard 4 kB-pages are used. Table III further shows the number of L1 cache and TLB misses for all three operations which we will use below to explain the runtime behavior for each operation as the input size increases.

We first explain why this effect occurs for $access(B, i)$. Performing $access(B, i)$ consist of: (1) address translation from the virtual address of $B[i]$ and (2) accessing the physical memory location

DOI: 10.1002/spe

of the bitvector corresponding to $B[i]$. For 4 kB-pages the 512 elements of the TLB can store translations of up to 512×4 kB $= 2$ MB of memory. For a 1 MB-bitvector all address translations can be performed using the TLB. After a few initial TLB misses for the first random accesses, no more TLB misses occur. The main cost is therefore the L2 miss which occures when fetching the bitvector data from L3 cache.

For bitvectors in the range from 2 MB to 12 MB, the bitvector still fits in L3 cache, but TLB misses occur as more memory pages are accessed than TLB entries are available. The handling of a TLB miss is cheap, since the page table for a 12 MB index is about a 12 kB and fits in the L1 cache. This is still the case for bitvectors in the range from 12 MB to 32 MB, but the bitvectors themselves cannot be completely held in L3 cache (as our L3 cache is 12 MB large) and therefore L3 misses occur and the bitvector data is transfered from RAM. In the range from 32 MB to 256 MB the page table is of size 32 kB to 256 kB and does not fit in L1 cache anymore. Therefore, each TLB miss now forces a L1 miss to fetch the page table entry from L2 cache. In the range from 256 MB to 12 GB, the page table is larger than the L2 cache and thus the $access$ operation can cause a L2 miss to update the TLB.

Finally, for bitvectors larger than 12 GB, the page table is larger than the L3 cache. Therefore looking up an address of one page table entry can now itself result in a TLB miss, which in turn can be handled by one L1 miss to access an entry of the upper level of the hierarchical page table. Accessing and loading this page table entry into memory causes another L3 miss. In total one $access$ operation can result in 2 TLB and 3 cache misses when 4 kB pages are used. This can be seen in Table III: For a 64 GB bitvector, the mean number of TLB misses per $access$ query is 1.9, and the mean number of L1 cache misses is 3.1.

Based on these findings, we now explain the runtime of RANK-V and RANK-1L. The cost of RANK-V are two memory accesses plus one $popcnt$ operation. For the 64 GB instance the TLB and L1 misses are two times the misses for one $access$ operation. We can observe in Figure 3 that the runtime is also doubled. Using the BW $popcnt$ adds extra overhead. The new RANK-1L structure performs only one memory access to the superblock and sequentially reads, for a block size of 256 bits, at most 32 additional bytes (equal to four 64-bit words). The latter may result in an extra L1 cache miss, since the blocks are not align to the 64 byte cache lines.

The use of 1 GB-pages reduces the number of TLB misses. For smaller instance sizes no TLB misses occur. For large bitvectors, the second TLB miss caused by the access to the large page table is no longer needed. We observe in Table III and Figure 3 that the transition from not having a TLB miss to having a TLB miss happens before the 1 GB size is reached. There are only 4 TLB entries for 1 GB-pages available in the TLB. We would therefore expect this to only occur at bitvectors of sizes larger than 4 GB. A reason for this might be that the operation system also stores the kernel itself in one hugepage, which in turn affects the number of TLB entries available to other programs. Further note that the runtime for the 64 GB bitvector increases slightly. As the data structure, including the 25% $rank$, is 80 GB, therefore larger than the memory of one NUMA node, which is 72 GB, an extra cost has to be paid for the RAM access.

Overall, the combination of the BLT and HP features results in a significant performance improvement for the $rank$ structures compared to the BW and no HP version. Our improvements allow a $rank(B, i, c)$ operation at the cost close to that of one memory access and are therefore almost optimal for this hardware configuration.

6.4. Select on Uncompressed Bitvectors

Next, we focus on $select(B, i, c)$. We first examine the properties of the binary search solutions SEL-BS, SEL-BSH, the 2-level solution SEL-C and the two select (SEL-V9 and SEL-VS) solutions proposed in [31] under two extreme environmental feature combinations: TBL+ 4 kB pages and BLT+ HP. We choose instance sizes of 16 MB and 16 GB to highlight the effects of address translation. For each instance we further use bitvectors of densities 5%, 20% and 50% as, unlike $rank$, the density can affect the performance of different $select$ data structures.

For each density we perform 10 million random $select$ queries and show the mean runtime performance per operation for densities 5% and 50% in Figure 4. For all 16 MB test instances,

DOI: 10.1002/spe

Table III. Average TLB and Level 1 cache misses for a single *rank* or *access* operation on uncompressed bitvectors averaged over 10 million queries for 4 kB and 1 GB pages.

	TLB misses / L1 cache misses per operation							
	1 MB	16 MB	64 MB	256 MB	1 GB	4 GB	16 GB	64 GB
4 kB pages (no HP)								
RANK-V	0.0/2.0	1.6/3.1	1.9/3.8	1.9/4.0	2.1/4.3	2.8/5.0	3.6/5.7	3.9/6.0
RANK-1L	0.0/2.0	0.9/2.3	0.9/2.0	1.0/3.0	1.0/3.2	1.2/3.7	1.3/4.0	1.9/4.3
BV *access*	0.0/1.0	0.8/1.6	0.9/2.0	1.0/2.0	1.0/2.1	1.4/2.6	1.8/2.9	1.9/3.1
1 GB pages (HP)								
RANK-V	0.0/2.0	0.0/2.1	0.3/2.1	1.4/2.1	1.8/2.1	2.0/2.1	2.0/2.1	2.0/2.1
RANK-1L	0.0/2.0	0.0/2.0	0.2/2.0	0.9/2.0	1.0/2.0	1.1/2.0	1.1/2.0	1.1/2.0
BV *access*	0.0/1.0	0.0/1.1	0.0/0.9	0.7/1.1	0.9/1.1	0.9/1.1	1.0/1.1	1.0/1.1

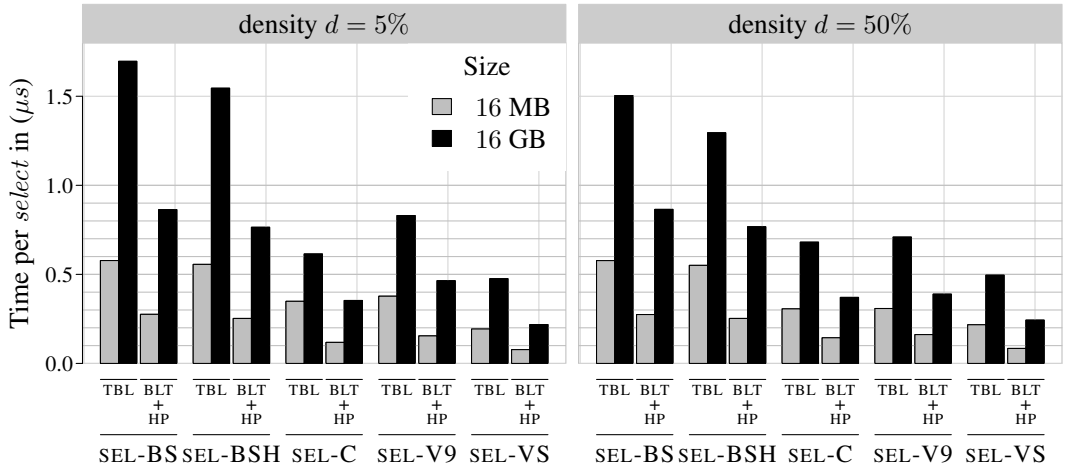

Figure 4. Average time for a single *select* operation on uncompressed bitvectors dependent on the size (16 MB/16 GB), the density (5/50) of the input, the implementation (SEL-BS/SEL-BSH/SEL-C/SEL-V9/SEL-VS) and the used environmental features (TBL/BLT+HP).

the bitvector and the support structure fit in one 1 GB hugepage. Therefore, all structures benefit from enabling the features (BLT+HP). All structures roughly become twice as fast for the small test case. Overall, the two level structure SEL-C outperforms SEL-BS, SEL-BSH and SEL-V9 for small instances, but is slower than SEL-VS. Further note that the performance of SEL-BSH and SEL-BS is almost identical as address translation for small instances does not significantly affect runtime performance (see our analysis of *rank* for details).

For the 16 GB instances, all runtimes increase due to the increasing cost of TLB and cache misses as described in the discussion of our *rank* experiment in Section 6.3. As expected SEL-BSH outperforms SEL-BS since the first 10 memory accesses cause no TLB miss. However, the unoptimized implementation (TBL+ 4 kB pages) of our 2-level SEL-C approach outperforms both binary search approaches even with BLT+ HP enabled. Without the features, SEL-C is slightly faster than SEL-V9 for $d = 5\%$ and roughly the same speed for $d = 50\%$. For all instances SEL-VS outperforms our new two level approach SEL-C.

Next we measure the mean number of TLB and L1 cache misses per operation in the experiment. The results are depicted in Table IV. Note that we do not include SEL-V9 in this discussion as it is outperformed by SEL-VS and SEL-C which both use significantly less space (see Figure 5). As expected, for the large test instances, the binary search methods procude significantly more TLB misses as SEL-C and SEL-VS. Without hugepages, the effect of the binary search "hints" can clearly be observed as the number of TLB misses is roughly reduced by 15. Hugepages reduces the effect

DOI: 10.1002/spe

Table IV. Average number of TLB misses/L1 cache misses per select operation dependent on implementation (SEL-BS/SEL-BSH/SEL-C/SEL-VS), $select_{64}$ method (TBL/BLT) and activation of 1 GB pages (HP) on bitvectors of different sizes (16 MB/16 GB) and densities (5/20/50). The two columns on the right contain the space overhead of the data structures in percent of the original bitvector size.

	TLB misses / L1 cache misses per operation				overhead in %	
	SEL-BS		SEL-BSH		SEL-BS	SEL-BSH
	TBL	BLT+HP	TBL	BLT+HP		
density $d = 5$						
16 MB	1.2 / 13.8	0.0 / 12.5	1.3 / 10.9	0.0 / 10.1	19	19
16 GB	25.6 / 67.9	7.4 / 68.7	10.3 / 43.3	5.9 / 40.3	20	19
density $d = 20$						
16 MB	1.2 / 14.2	0.0 / 13.8	1.3 / 11.1	0.0 / 10.9	19	19
16 GB	25.6 / 68.4	7.4 / 68.8	10.3 / 43.5	6.0 / 40.2	20	20
density $d = 50$						
16 MB	1.3 / 13.8	0.1 / 13.2	1.5 / 10.6	0.1 / 10.6	19	19
16 GB	25.7 / 68.0	7.5 / 69.0	10.6 / 44.0	6.6 / 40.6	20	20
	SEL-C		SEL-VS		SEL-C	SEL-VS
	TBL	BLT+HP	TBL	BLT+HP		
density $d = 5$						
16 MB	0.9 / 7.9	0.0 / 7.5	1.3 / 4.3	0.0 / 3.6	2	11
16 GB	4.9 / 13.7	3.9 / 10.2	3.0 / 7.8	2.8 / 3.6	2	11
density $d = 20$						
16 MB	1.3 / 7.3	0.0 / 6.7	1.4 / 4.5	0.0 / 3.7	5	11
16 GB	5.0 / 13.0	4.4 / 8.9	3.0 / 8.1	2.9 / 3.8	5	11
density $d = 50$						
16 MB	2.1 / 6.9	0.0 / 5.7	1.2 / 5.2	0.0 / 4.3	12	7
16 GB	5.2 / 12.9	4.9 / 8.1	3.0 / 8.8	2.8 / 4.4	12	7

of the hints as more steps of the binary search procedure can be performed on a single page. For 16 GB, we can further observe that for SEL-C and BLT+HP the number of TLB misses increases for bitvectors of higher densities while, at the same time, the number of L1 misses decreases. The TLB misses increase since, for higher densities, the index part of the data structure is larger, while the average block size decreases. For $d = 50\%$ the final block is 128 bits or 16 bytes large whereas we require 1280 bits or 160 bytes for $d = 5\%$. Therefore, we load two cache lines for $d = 50\%$ while we only require one cache line for the $d = 5\%$. This explains the difference of two L1 misses (10.2 compared to 8.1) shown in Table IV for 16 GB and densities $d = 5$ and $d = 50$. The TLB and L1 cache miss performance of SEL-VS is better than SEL-C. This is caused by the way both structures store bit positions. SEL-VS uses only byte aligned accesses to words of size 16 or 64 bits. SEL-C stores positions bit-compressed. This results in slower runtime performance and an increased number of L1 cache misses.

Table IV further shows the space overhead required for each select structure. The binary search methods use 20% overhead. The size of the "hints" in SEL-BSH is only 8 kB and is therefore negligible. For densities $d = 5$ and $d = 20$ SEL-C is significantly smaller than SEL-VS. However, for $d = 50$ SEL-VS is smaller than SEL-C.

As suggested by [31], it is important to evaluate the performance of select structures on uneven distributed bitvectors. We therefore evaluate the performance of all select structures discussed above on several real world data sets. The "real world" bitvector instances are extracted from the Huffman-shaped wavelet tree as described in Section 6.1. Figure 5 shows the time-space trade-offs of each structure for all test instances. We contrast the mean time per *select* operations over 10 million random queries with the space overhead of each structure in percent of the original bitvector. Note that the real world data sets roughly have about 50% density as they represent a Huffman-shaped

DOI: 10.1002/spe

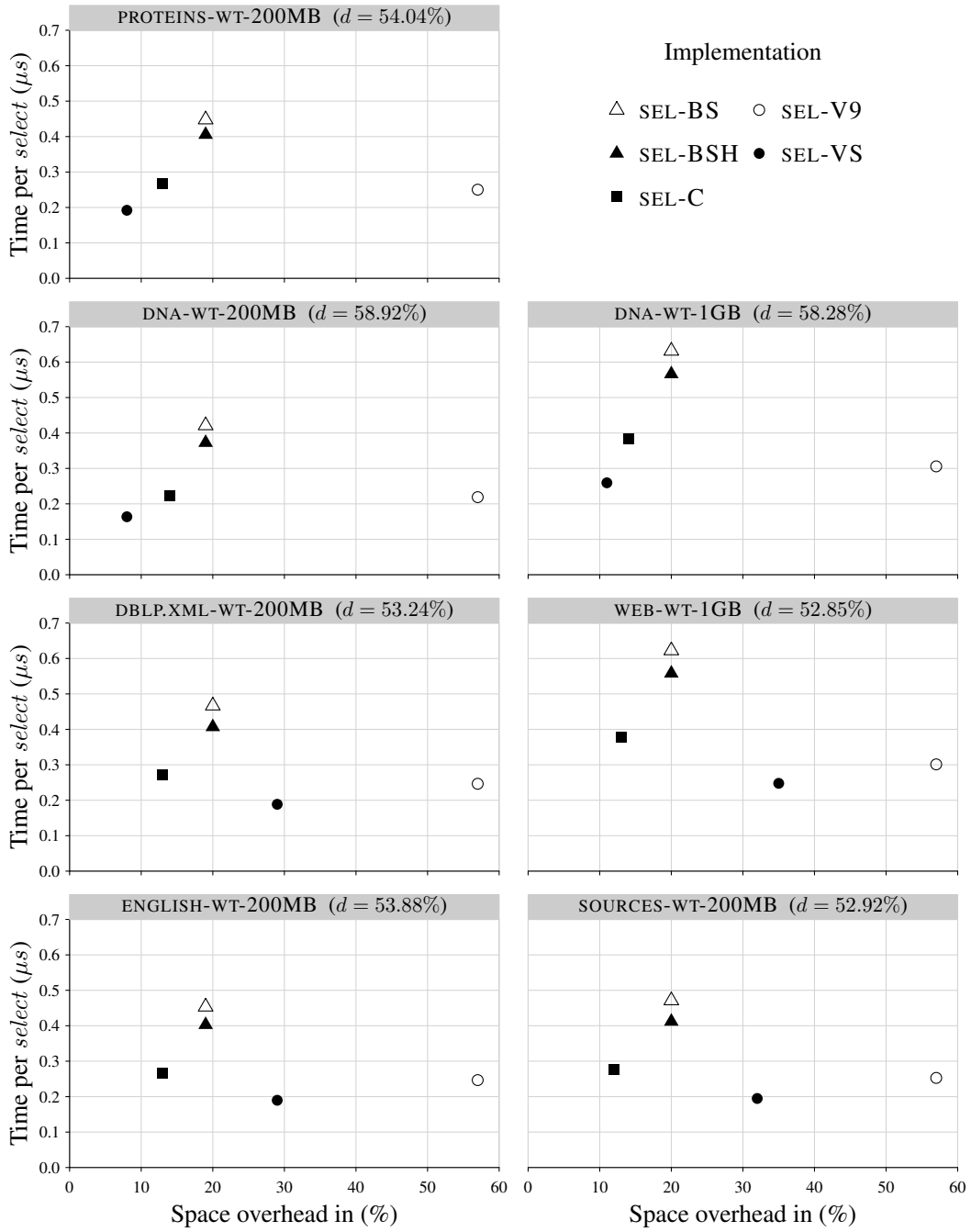

Figure 5. Time-space trade-offs for a single *select* operation on uncompressed bitvectors for different "real world" data sets of the implementations SEL-BS, SEL-BSH, SEL-C, SEL-V9 and SEL-VS. For all implementations the features BLT+HP were used.

wavelet tree. Another property is that they contain long runs as well as evenly distributed regions. Long runs occur frequently in the bitvectors of human generated texts, which contains words and are structured. We observe in Figure 5 that overall SEL-BS and SEL-BSH are not competitive. SEL-V9 is faster than the binary search methods but requires 57% overhead. SEL-VS is always the fastest method but the space usage varies between different data sets. For WEB-WT-1GB, SEL-VS uses

DOI: 10.1002/spe

35% overhead whereas SEL-C uses only 13%. The same behavior can be observed for SOURCES-WT-200MB, ENGLISH-WT-200MB and DBLP.XML-WT-200MB. On the one hand, for bitvectors with few runs (PROTEINS-WT-200MB, DNA-WT-200MB,DNA-WT-1GB) the space usage of SEL-VS is about the same as for evenly distributed bitvectors and therefore less than SEL-C.

We independently evaluate the effect of different $select_{64}$ implementations on the *select* structures. For all evaluated *select* structures, only one $select_{64}$ operation is performed to determine the position of the i-th bit in a target word x. Before this step, potentially many other operations such as *popcnt*s in a sequential scan or binary search are performed to determine x. We measure the effect of using a slow $select_{64}$ method (TBL) compared the fastest method (BLT) of the target word on the overall performance of the structure. For small instances, the running time for SEL-C and SEL-VS decreases by 10 to 20%, for the binary search structures it decreased by 7%. For large instances, the running time for SEL-C and SEL-VS decreases only by 2% to 5%. The binary search solutions do not benefit from improved $select_{64}$ on large instances as the running time is dominated by binary search and the resulting TLB misses. Interestingly, the running time for SEL-V9 decreases by 30% for all instances while all other *select* structures improve mostly for small instances.

Another consideration when choosing a *select* structure is construction cost. We use both *popcnt* and $select_{64}$ methods during the construction of our SEL-C structure. We observed an improvement in construction cost by up to an order of magnitude (up to 30 times faster for the random bitvectors used in our experiments) compared to versions with simple bit-by-bit processing during construction.

6.5. Rank and Select on Compressed Bitvectors

We now turn our attention to the H_0-compressed bitvector representation. We have seen in Figure 2 that this representation can lead to sigificant space savings. In the following we explore how the runtime is affected by different choices of the block size K.

In the implementation of BV-R^3K we used built-in 64- and 128-bit integers for block sizes $K \leq 64$ and $K \leq 128$. For $K \geq 129$ we used our own tailored class for 256-bit integers. Note that the size of the lookup tables for Pascal's triangle for the on-the-fly decoding is therefore 32 kB, 256 kB and 2 MB for the three different integer types. In the special case of $K = 15$ we do not use on-the-fly decoding but use on access to a lookup table of size 64 kB to retrieve the 15-bit block instead. In this case we also use broadword computing to calculate the sum of multiple block types $\kappa_i s$. For all other block sizes we sum up each κ_i individually and use the on-the-fly decoding in the last block $b_{i'}$.

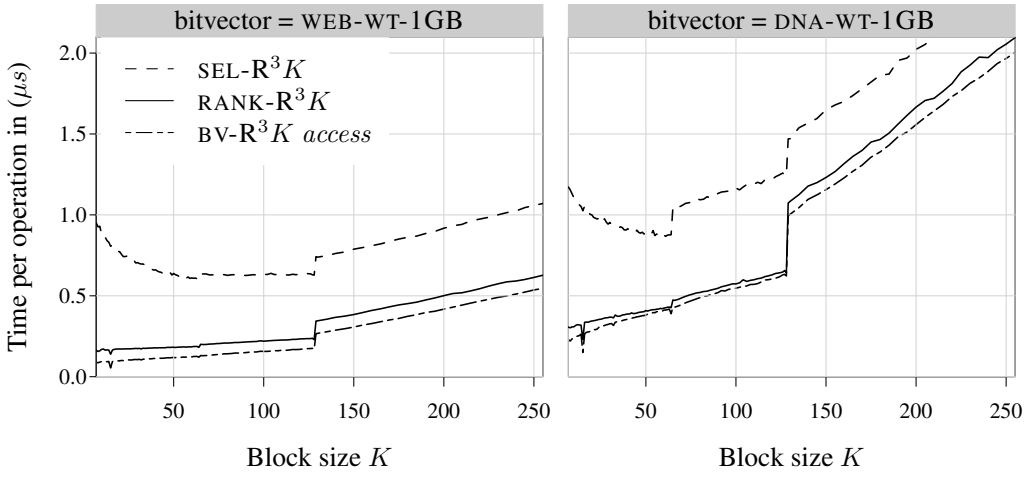

Figure 6. Query times for the operations on the H_0-compressed bitvector as function of block size K. The sample rate t was set to 32.

DOI: 10.1002/spe

Figure 6 depicts the resulting runtime for the three operations $access$, $rank$, and $select$ as function of K for the bitvectors originating from text indexing. We first concentrate on the operations $access$ and $rank$. Note that the specialized implementation for $K = 15$ can be recognized clearly: its is about twice as fast as the on-the-fly decompression with comparable block sizes. In fact additional experiments showed that our solution for this special case is slightly faster (about 10-30%) than the original implementation of Claude and Navarro [3] for $rank$ and twice as fast for $access$.

We can observe that the runtime of the on-thy-fly decoding version linearly depends on the block size. The constant of the linear correlation is determined by two factors: the integer type used and the structure of the original bitvector, where the latter has a stronger impact then the former. The reason for that is the number of uniform blocks in WEB-WT-1GB is much larger than in DNA-WT-1GB, e.g. for $K = 63$ it is 84% compared to 28%, thus we actually do less decoding in WEB-WT-1GB. The percentage of affected blocks depending on K is shown in Figure 7. Up to 90% of all blocks in WEB-WT-1GB are uniform and can therefore be decoded efficiently. As shown in Figure 7, this percentage is much smaller for DNA-WT-1GB.

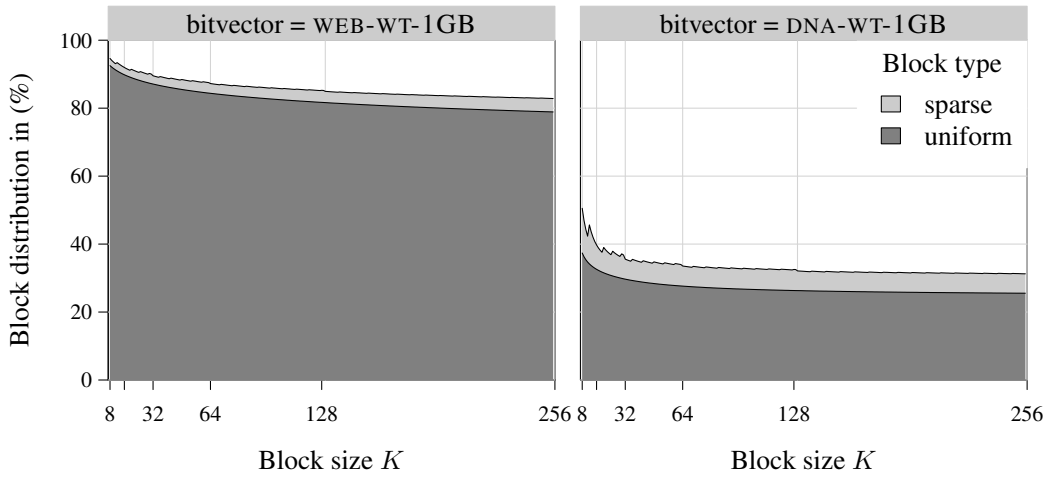

Figure 7. Percentage of blocks in a H_0 compressed bitvector representation which can be optimized using technique one (uniform blocks) and technique two (sparse blocks) for wavelet trees of files WEB, DNA

The described optimization for $access$ results in a significant speed-up: $access$ is two times faster than $rank$ for K and a high ratio of uniform blocks.

The time for $select$, which is implemented using binary search, slightly decreases when we increase the block size and native arithmetic is used. For $K \geq 64$ the decoding costs become too dominant and we cannot observe the reduced time for the binary search on less rank samples any more. The effect of the integer type can be clearly observed by the transitions from using 64- or 128-bit integers to the custom 256-bit integers.

Our implementation decodes only as many bits as necessary to answer the specific operation as proposed in [24]. Further experiments show that this optimization reduces the running time by 50%. The runtime was again reduced by 50% by applying the first optimization proposed in Section 5 for bitvectors from the indexing domain. The second proposed optimization – decoding with binary searching Pascal's triangle if only a few bits are set in the block – only improved the runtime for the random bitvectors of densities $d \leq 5\%$. The runtime was reduced by 10% for $d = 5\%$ and 50% for $d = 1\%$.

6.6. Effects of Improved rank/select on FM-Indexes

Next we evaluate how our improvements affect the performance of different types of succinct text indexes. We are especially interested in the effects caused by using faster $rank$ and $select$ implementations on bitvectors to the performance of $count$ queries. A $count$ query for a pattern of

DOI: 10.1002/spe

length m can be reduced to about $2mH_0$ *rank* queries if a Huffman-shaped wavelet trees is used, or at most $2m(H_0 + 2)$ *rank* and $2m$ *select* queries if a run-length encoded wavelet tress is used [18]. We expect that the improvements of basic data structures directly propagate to the FM-indexes, since our benchmarking methodology in the previous sections and performing a *count* query are similar. Both require the execution of a series of consecutive *rank* and/or *select* operations on bitvectors.

Operations *locate* and *extract* are more complex. *locate* additionally depends on the sampling of suffix array SA and *extract* on the sampling of the inverse SA (ISA). We also expect improvements for these operations as they both depend on performing *rank* and/or *select* operations.

Baseline Implementations Ferragina et al. [6] provide a set of highly optimized FM-indexes. They also provide all resources of their paper (including the source code) as part of the *Pizza&Chili* corpus discussed in Section 6.1. In the following we will compare our work to these implementations to ensure that our indexes are competitive to commonly used baseline implementations. Here is a short description of the baselines implementations used:

- SSA — The Succinct Suffix Array [18] is an FM-index based on a Huffman-shaped wavelet tree. The wavelet tree uses uncompressed bitvectors supported by one-level rank data structures using 5% space overhead. The *popcnt* implementation is TBL; authors are Veli Mäkinen and Rodrigo González.
- SSA-RRR — Same index as SSA but the H_0-compressed bitvector representation of [27, 28] implemented by Francisco Claude and Gonzalo Navarro [3] is used.
- RLFM — Run-length wavelet tree [18] implementation by Veli Mäkinen and Rodrigo González. The two indicator bitvectors for the run heads in the BWT are represented by uncompressed bitvectors and the same rank structure as in SSA. The *select* operation is solved by a binary search over the rank samples.
- CSA — The Compressed Suffix Array [29] (CSA) is not based on the backward decoding approach but on the forward decoding approach. This approach is based on an array Ψ, which can be stored in compressed form by employing delta and self-delimiting coding. Fast access is achieved by adding samples. Note that Sadakanes' implementation is not depended on *rank* and *select* structures.
- SAu — Plain suffix array [20] implementation of Veli Mäkinen and Rodrigo González.

We compiled the 32-bit implementations with all compiler optimizations and added the flag `-m32` since we use our 64-bit platform. Note that this limits the usage of theses indexes to small inputs.

Corresponding sdsl Implementations We create comparable sdsl indexes for the presented baseline indexes by parametrizing sdsl FM-indexes and CSAs with comparable basic data structures. The following list gives an overview.

- FM-HF-V5 — FM-index based on a Huffman-shaped wavelet tree (`wt_huff`) which is parametrized with the uncompressed bitvector (`bit_vector`) and the 6.25% overhead rank structure `rank_support_v5<>`. Corresponds to the baseline index SSA.
- FM-HF-R^{3}15 — Same as FM-HF-V5 except that the wavelet tree is parametrized with the compressed bitvector `rrr_vector<15>` and its associated rank and select structures. Corresponds to the baseline index SSA-RRR.
- FM-HF-R^3K — A family of more space-efficient FM-indexes. Realized by parametrizing the wavelet tree of FM-HF-V5 by the `rrr_vector<`K`>`
- FM-RLMN — FM-index based on a run-length compressed wavelet tree (`wt_rlmn`) which is configured with compressed bitvectors (BV-SD) for the two indicator bitvectors. Corresponds to the baseline index RLFM.
- CSA-SADA — The sdsl CSA class (`csa_sada`) parametrized with Elias-δ coder and Ψ sampling density based of 128. Corresponds to CSA.

Note that all these indexes do not contain SA or ISA samples which are not required to answer *count* queries efficiently.

DOI: 10.1002/spe

Table V. Space and time performance of four *Pizza&Chili* FM-index implementations on five different inputs for count queries. The space of an index is stated in percent of the input text. The time is the average time to match one character in a count query. Times were determined by executing $50,000$ queries, each for patterns of length 20, which were extracted from the corresponding texts at random positions.

	SSA		SSA-RRR		RLFM		SAu	
200 MB test instance	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)
DBLP.XML	1.300	69	1.776	34	6.108	52	0.436	500
DNA	0.548	29	0.812	30	1.216	62	0.388	500
ENGLISH	1.156	60	1.676	40	1.744	67	0.372	500
PROTEINS	1.024	56	1.668	55	1.704	76	0.368	500
SOURCES	1.356	72	1.960	41	1.864	61	0.364	500

Table VI. Space and time performance of four sdsl FM-index implementations. The experiment was the same as in Table V. All implementations in this experiment use method TBL for $popcnt$.

	FM-HF-V5		FM-HF-R^{3}15		FM-RLMN		FM-HF-R^{3}63	
200 MB test instance	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)	Time (μs)	Space (%)
DBLP.XML	1.096	70	1.316	32	3.456	34	2.048	17
DNA	0.404	29	0.728	28	1.484	79	1.660	24
ENGLISH	0.976	61	1.472	38	2.272	69	2.648	27
PROTEINS	0.872	56	1.604	53	2.128	89	3.228	48
SOURCES	1.208	73	1.688	39	2.460	53	2.820	26

Operation $count$ Without Optimizations The performance of $count$ was measured by recreating the experiment of Ferragina et al [6] (Table VI in their work). Table V contains the result of their indexes on our hardware. A nice observation is that the times are almost halved compared to the results in [6] on older hardware. Table VI shows the resulting index sizes and runtime for the sdsl implementations. Note that we use TBL for $popcnt$ without activating hugepages to ensure a fair comparison to the baseline implementations. As expected the size of SSA and FM-HF-V5 are almost identical since the rank structure size only differs by 1.5%. Our FM-HF-V5 index is slightly faster than SSA since the 5% overhead $rank$ implementation used in SSA scans blocks 66% larger than that of RANK-V5. The space of FM-HF-R^{3}15 is slightly smaller than that of SSA-RRR since we use $\log n$ bits to store an integer instead of a computer word. Our implementation is again faster as we use our algorithmic optimizations described in Section 5. Our implementation FM-RLMN, which uses the compressed bitvector representation, is smaller than RLFM for two test cases (DBLP.XML, SOURCES). It is however larger for all other test cases. We attribute this to the high compressibility of the other test cases. Our compressed bitvector representation (BV-SD) is engineered to compress *sparse* bitvectors efficiently. Dense bitvectors, which occur in an FM-Index for highly compressible data due to the large number of runs in the Burrow-Wheeler-Transform, can however not be compressed efficiently. Furthermore, the compressed bitvector causes a slowdown. Unexpectedly, FM-RLMN is two times faster than RLFM on the XML-instance.

The last column of Table VI further shows the results of using larger block size $K = 63$ to achieve better compression of BV-R^3K in FM-HF-R^3K. As expected the space is significantly reduced for the compressible texts and the runtime doubles compared to FM-HF-R^{3}15. Recalling Figure 6, we would expect an even greater slowdown. However, $rank$ queries on the lower levels of the wavelet tree have a better locality of reference. Therefore the performance of the H_0 compressed bitvectors shown in Figure 6 can only be used to predict the runtime of the $rank$ queries in the upper levels where rank positions are more random.

Overall our implementations are competitive and tend to outperform the ones used by Ferragina et al. which are available through the *Pizza&Chili* corpus.

DOI: 10.1002/spe

Table VII. Time performance of the sdsl FM-index implementations of Table VI dependent on the *popcnt* method used (see Section 6.2) and the usage of 1 GB pages (HP) to avoid TLB misses. The time difference is stated as relative difference to the corresponding implementation which uses method TBL for *popcnt*.

	FM-HF-V5		FM-HF-R^{3}15		FM-RLMN		FM-HF-R^{3}63	
	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)
200 MB test instance								
DBLP.XML								
TBL	1.096		1.316		3.456		2.048	
BW	0.924	−16	1.324	+1	2.600	−25	2.064	+1
BLT	0.764	−30	1.304	−1	1.876	−46	2.036	−1
BLT+HP	0.644	−41	1.140	−13	1.764	−49	1.924	−6
DNA								
TBL	0.404		0.728		1.484		1.660	
BW	0.348	−14	0.724	−1	1.408	−5	1.652	+0
BLT	0.280	−31	0.716	−2	1.248	−16	1.660	+0
BLT+HP	0.252	−38	0.628	−14	1.356	−9	1.688	+2
ENGLISH								
TBL	0.976		1.472		2.272		2.648	
BW	0.832	−15	1.480	+1	2.108	−7	2.644	+0
BLT	0.688	−30	1.472	+0	1.900	−16	2.660	+0
BLT+HP	0.600	−39	1.300	−12	2.200	−3	2.756	+4
PROTEINS								
TBL	0.872		1.604		2.128		3.228	
BW	0.752	−14	1.596	+0	1.980	−7	3.264	+1
BLT	0.624	−28	1.560	−3	1.824	−14	3.244	+0
BLT+HP	0.552	−37	1.408	−12	2.100	−1	3.432	+6
SOURCES								
TBL	1.208		1.688		2.460		2.820	
BW	1.044	−14	1.680	+0	2.196	−11	2.804	−1
BLT	0.856	−29	1.656	−2	1.996	−19	2.824	+0
BLT+HP	0.744	−38	1.476	−13	2.276	−7	2.652	−6

Operation *count* With Optimizations We now study the effect of our environmental features and optimizations on the performance of our indexes. We perform the same experiment as above while varying the applied features discussed in Section 6.1. Table VII shows the results of the experiment. The runtime of FM-HF-V5 is reduced by about 40% for all test cases when BLT and HP are activated. Our implementation is twice as fast as the highly optimized *Pizza&Chili* counterpart. Surprisingly, for the DNA-instance FM-HF-V5 outperforms the plain uncompressed suffix array solution which takes 17 times the space of FM-HF-V5 (see Table V column SAu). This is especially interesting as succinct data structures tend to be slower than their uncompressed counterpart. The effect on the Huffman-shaped wavelet tree based FM-indexes using BV-R^3K is not significant. For both $K = 15$ and $K = 63$ scanning the array κ and decoding of the original blocks b_i of the underlying compressed bitvector contributes the most towards the running time of FM-HF-R^3K and can thus not be improved by BLT or hugepages. Our second compressed solution, FM-RLMN, profits mostly from the BLT feature for two reasons: first, the *rank* operations on the underlying Huffman-shaped wavelet tree on uncompressed bitvectors is accelerated and second the to *rank* queries in BV-SD, which translate to *select* queries on SEL-C, are faster.

We rerun the experiment of Table VII on our second experimental machine, to verify our observations. Since Mac OS X 10.7.3 does not support 1 GB pages we only show results for the first three environmental features. The results are depicted in Table VIII. In most cases the runtime for the basic *popcnt* implementation decreases slightly compared to the results on the main experimental machine. This is expected since the maximum clock speed of CPU on the MacBook is 3.00 GHz compared to 2.93 GHz on the server. The impact of applying BW and BLT is roughly the same and the final query time of FM-HF-V5 is again twice as fast as the corresponding *Pizza&Chili* implementation.

DOI: 10.1002/spe

Table VIII. Result of the experiment of Table VII conducted on the MacBook.

	FM-HF-V5		FM-HF-R^{3}15		FM-RLMN		FM-HF-R^{3}63	
	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)	Time t (μs)	Δt (%)
200 MB test instance								
DBLP.XML								
TBL	1.071		1.353		3.133		2.123	
BW	0.903	−16	1.359	+0	2.644	−16	2.110	−1
BLT	0.775	−28	1.350	+0	1.989	−37	2.125	+0
DNA								
TBL	0.368		0.705		1.463		1.550	
BW	0.317	−14	0.682	−3	1.375	−6	1.556	+0
BLT	0.279	−24	0.676	−4	1.247	−15	1.532	−1
ENGLISH								
TBL	0.875		1.392		2.212		2.587	
BW	0.757	−13	1.386	+0	2.068	−7	2.598	+0
BLT	0.653	−25	1.363	−2	1.941	−12	2.640	+2
PROTEINS								
TBL	0.807		1.463		1.972		3.192	
BW	0.673	−17	1.458	+0	1.939	−2	3.180	+0
BLT	0.595	−26	1.482	+1	1.841	−7	3.207	+0
SOURCES								
TBL	1.090		1.630		2.331		2.795	
BW	0.958	−12	1.575	−3	2.329	+0	2.799	+0
BLT	0.829	−24	1.555	−5	2.053	−12	2.812	+1

Operation *locate* **With Optimizations** As mentioned above, the *locate* operation is more complex than a *count* query. In general, *locate* is implemented by storing samples of SA. Each value $\mathsf{SA}[i]$ can be computed by iteratively calculating the positions of the next longer suffix until a sampled suffix $\mathsf{SA}[j]$ is reached. The calculation of the next longer suffix can be done by one *rank* query on a wavelet tree and we refer to it as calculating the LF-function. In case of CSAs, the inverse of LF, called Ψ, is stored in compressed form. Ψ returns the next shorter suffix and is applied until a sampled suffix is reached. In both cases we know the number of iterations k and can therefore return $\mathsf{SA}[i] = \mathsf{SA}[j] + k$ respectively $\mathsf{SA}[i] = \mathsf{SA}[j] - k$.

There exists two simple strategies to sample SA. Let s_{SA} be the SA sampling parameter. In the fist variant $\mathsf{SA}[j]$ is sampled when $\mathsf{SA}[j] \bmod s_{\mathsf{SA}} \equiv 0$, in the second when $j \bmod 0$. The first variant is generally referred to as *suffix array order sampling* whereas the second method is referred to as *text order sampling*. In the first strategy a bitvector marks the sampled positions j and a rank data structure calculates the index of the correspond sample in a satellite array. This adds space but it is guaranteed that LF is not performed more than $s_{\mathsf{SA}} - 1$ times to retrieve $\mathsf{SA}[i]$. The second version does not require an additional bitvector but does not guarantee running time bounded by the sampling rate. However, it has proven to work well on many inputs, including the test cases of the *Pizza&Chili* corpus. CSA uses text order sampling whereas all other *Pizza&Chili* indexes use an uncompressed bitvector to implement suffix array order sampling. We parametrized our implementations with both strategies and empirically determined that the suffix array order sampling was up to two times faster than text order sampling. Therefore, we opted to use the faster variant in the following experiment. The *Pizza&Chili* indexes use an uncompressed bitvector representation to mark the sampled position. In our indexes we use a compressed bitvector (BV-SD) to save space instead.

Figure 8 shows the results of the recreated *locate* experiment of Ferragina et al. (Figure 4 in [6]) including our solutions with all optimizations enabled. We varied s_{SA} in $\{4, 8, 16, 32, 64, 128, 256\}$ to compare indexes of different size and speed identical to the original experiment of Ferragina et al. In all but CSA, the sampling parameter for the ISA values was set to $s_{\mathsf{ISA}} = s_{\mathsf{SA}}$. Therefore we get a space overhead of $(2n\lceil \log n \rceil)/s_{\mathsf{SA}}$ bits for the samples. CSA sets $s_{\mathsf{ISA}} = 16 \cdot s_{\mathsf{SA}}$ and is therefore

DOI: 10.1002/spe

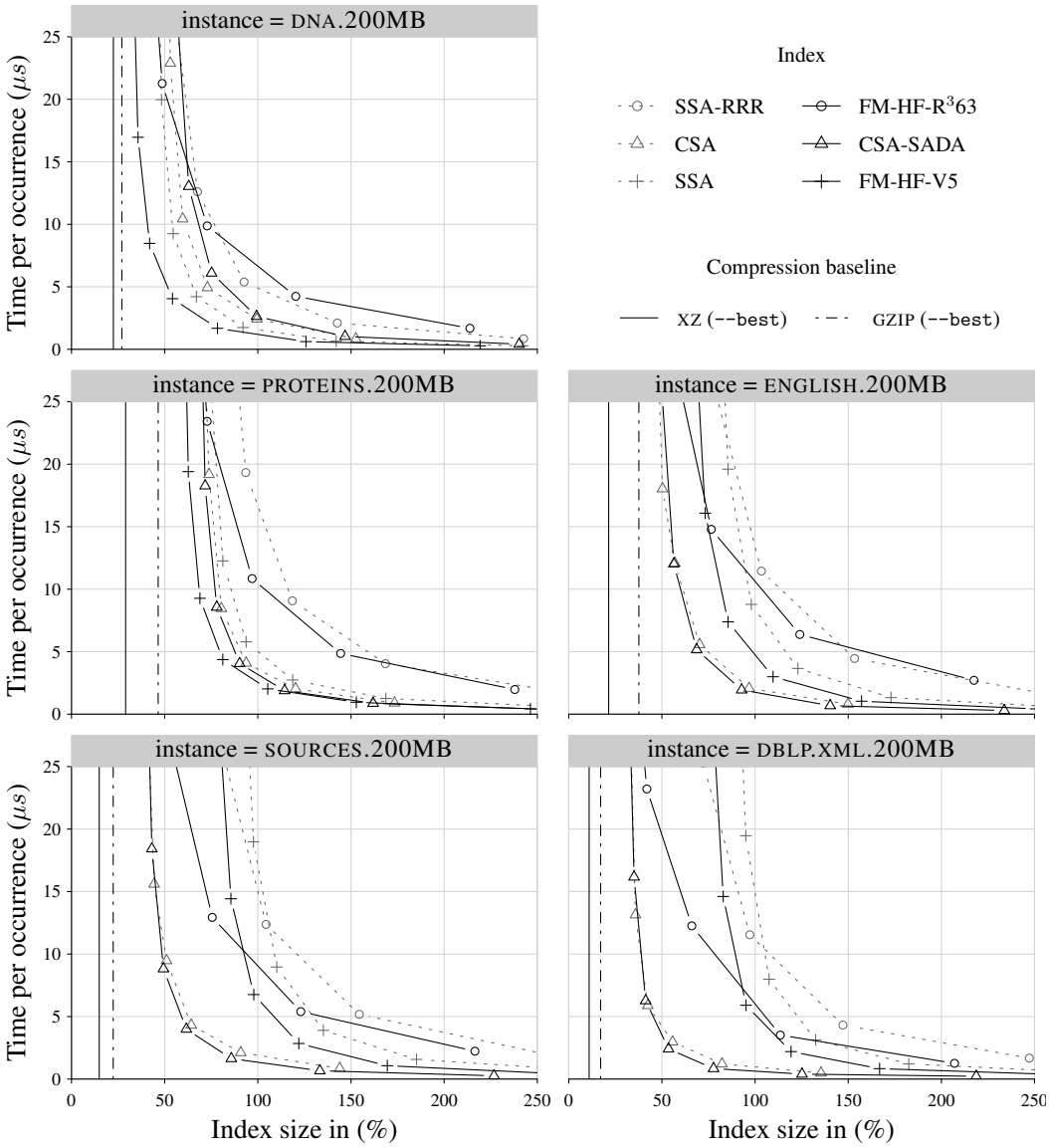

Figure 8. Time-space trade-offs for *locate* of different *Pizza&Chili* and sdsl indexes. Features BLT+HP were enabled for all sdsl indexes.

smaller than its counterpart CSA-SADA. The runtime of both implementations is about the same. This was expected, since CSA-SADA is not based on the optimized basic data structures,

However, we can observe significant improvements to the performance of FM-indexes based on wavelet trees. FM-HF-V uses less space than SSA since BV-SD is used to mark sampled SA positions. SSA spends about 20% space while FM-HF-V spends about 4% for $s_{\mathsf{SA}} = 32$. Using BV-SD to detect the samples is slower than using an uncompressed bitvector in SSA, but overall FM-HF-V is still faster, since we use faster rank structures to calculate LF. The best example for this behavior is the DNA.200MB case. While SSA requires about $20\mu s$ to calculate an occurrence at a compression rate of 50%, FM-HF-V only needs about $5\mu s$.

Last, we compare SSA-RRR and FM-HF-R^{3}63 which use compressed bitvectors to calculate LF. Note that we deliberately chose not to compare SSA-RRR with FM-HF-R^{3}15. We show, while FM-HF-R^{3}63 uses a slower bitvector representation, it still achieves a better time-space trade-off.

DOI: 10.1002/spe

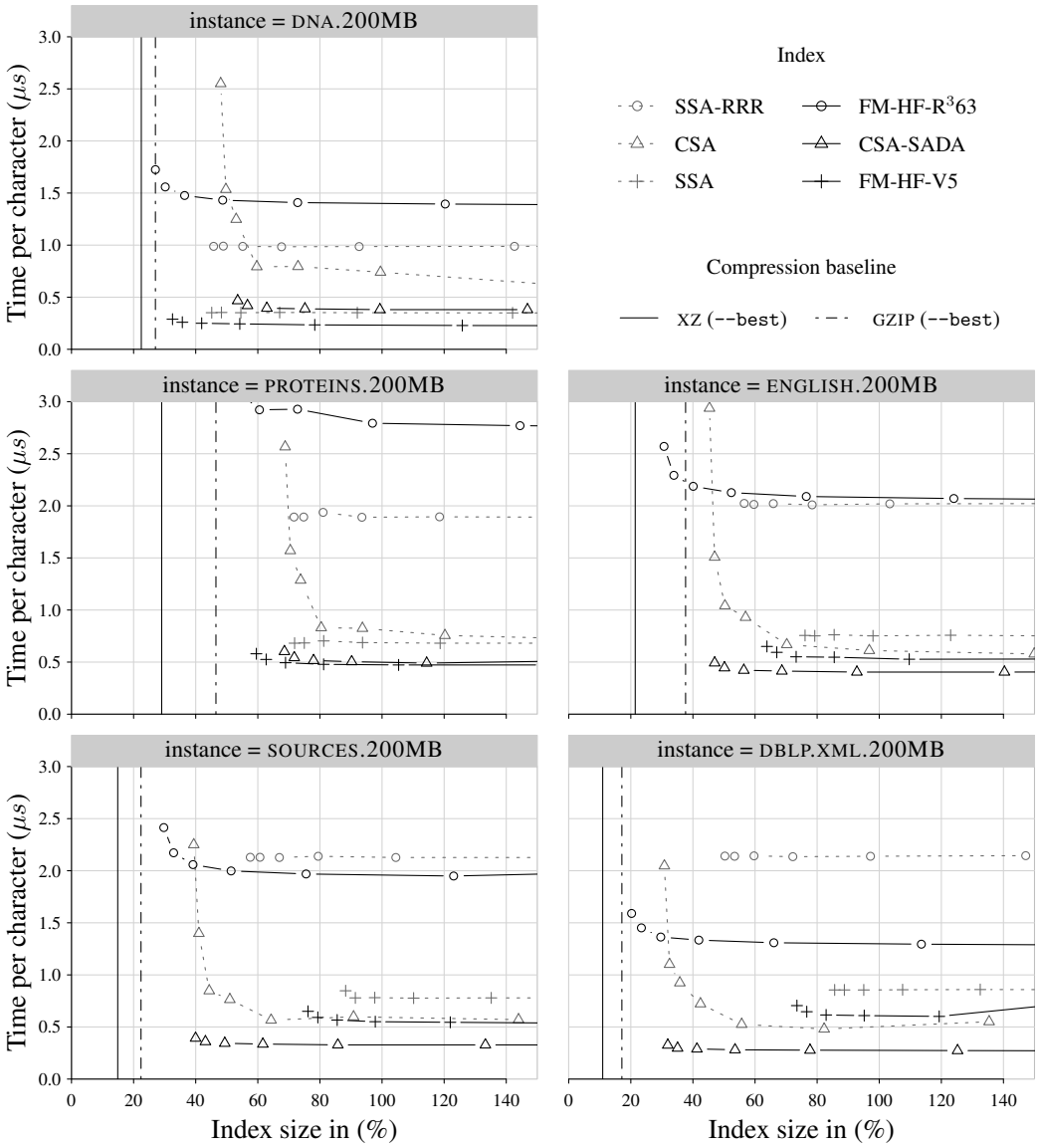

Figure 9. Time-space trade-offs for *extract* of different *Pizza&Chili* and sdsl indexes. Features BLT+HP were enabled for all sdsl indexes.

As expected the trade-off improves with test cases that are more compressible which we already analyzed in Figures 2 and 6.

Operation *extract* **with Optimizations** Extracting text $T[i..j]$ from a compressed index is performed again with a two step process. In the case of FM-indexes, the position of suffix j in SA is determined by calculating $p = \mathsf{ISA}[j]$. This is done in a similar way to the calculation of SA. Next, a table C of size σ, containing the index of the first suffix in SA prefixed by c, is used to translate p into the corresponding character $T[p]$. The process recovers the text backwards by decoding the preceding characters by applying LF. For CSAs the process is slightly changed: First the position of i in SA is determined and then Ψ is applied to decode the text in forward direction.

In this experiment we use the same index configuration as in the *locate* experiment to measure the performance of operation *extract*. Again we use the same experimental setup as [6]: we extract $10,000$ substrings consisting of 512 characters from the index. The results are depicted in Figure 9.

DOI: 10.1002/spe

Table IX. Count performance of FM-index implementations dependent on instance size and features. The indexes in the left part use compressed bitvectors and therefore take considerably less space then FM-HF-V and FM-HF-1L as shown in Figure 10.

	FM-HF-R^3K				FM-RLMN	FM-HF-V	FM-HF-1L
	$K = 15$	$K = 63$	$K = 127$	$K = 255$			
	Time (μs)	Time (μs)	Time (μs)	Time (μs)	Time (μs)	Time (μs)	Time (μs)
test instance							
WEB-64M							
TBL	1.183	2.085	3.509	13.864	2.362	0.789	0.843
BW	1.194	2.091	3.482	14.027	1.855	0.684	0.733
BLT	1.182	2.094	3.489	14.019	1.447	0.651	0.590
BLT+HP	1.008	1.991	3.393	13.714	1.316	0.519	0.508
WEB-64G							
TBL	3.488	3.675	4.794	17.571	8.237	2.440	2.684
BW	3.526	3.707	5.621	16.940	6.991	2.244	1.952
BLT	2.945	3.724	4.961	15.756	4.960	2.224	1.700
BLT+HP	1.780	2.796	4.353	15.496	4.177	1.085	0.870

As in the *count* experiment we first observe that the extraction speed for the *Pizza&Chili* indexes is much faster on our new hardware. Ferragina et al. [6] reported a maximal extraction speed of 1 MB/s. Now it is about 2.5 MB/s for DNA.200MB and about 2 MB/s for most other cases. The use of fast *popcnt* and the HP feature results in 4 MB/s for our FM-HF-V for the DNA case. In general the runtime of FM-HF-V is only about 60-70% the runtime of SSA. As in [6] the CSA dominates the runtime for the human generated texts. The high sampling rate in the CSA implementation is the main reason why the runtime degrades faster and is generally slower than that of CSA-SADA.

Finally, note that FM-HF-R^{3}63 reaches, or even breaks, the size of the GZIP compressed text and we are still able to decode between 0.3 MB/s and 0.6 MB/s.

Operation *count* on Massive Data Sets Finally, we evaluate the performance of our new *rank* data structure RANK-1L as well as our improvements to the implementation of *select* and H_0 compressed bitvectors on a massive data set. We show runtime performance (Table IX) for different feature sets as well as time-space trade-offs (Figure 10) for a small (64 MB) and a large (64 GB) data set. We further included – as in the previous experiments – two state-of-the-art compression baselines in the time-space trade-off graph to better evaluate the compression effectiveness of the different index types.

We parametrize the Huffman-shaped wavelet tree with the two 25% overhead rank structures RANK-V and RANK-1L and refer to the results as FM-HF-V and FM-HF-1L. We further also evaluate FM-index types using RANK-R^3K with $K = 15, 63, 127, 255$. These indexes are referred to as FM-HF-R^3K. We use prefixes of WEB-64GB and again use the same methodology as Ferragina et al's study [6]. We report the average query time per character of a *count* query for 50,000 patterns of length 20 which are extracted from random positions of the corresponding input prior to the experiment.

The results of the experiments are shown in Table IX and Figure 10. For FM-HF-R^3K a performance improvement caused by applying optimizations is only achieved for the large test instance. Enabling BLT for $K = 15$ results in faster runtime performance, since a shift and *popcnt* is used to determine the rank to the right offset, after a block b'_i is decoded by a table lookup. The on-the-fly decoding based solution with $K \neq 15$ stops decoding when the offset is reached and therefore does not profit from BLT. All FM-HF-R^3K type indexes profit from HP, but the effect decreases as K becomes larger and decoding the blocks inside the compressed bitvector dominates runtime. Note that for the 64 MB inputs, the size of the indexes is at most 30% (see Figure 10) of the original data set and the resulting page table is about 20 kB large which fits in the L1 cache. Therefore TLB handling is cheap and HP has almost no effect.

DOI: 10.1002/spe

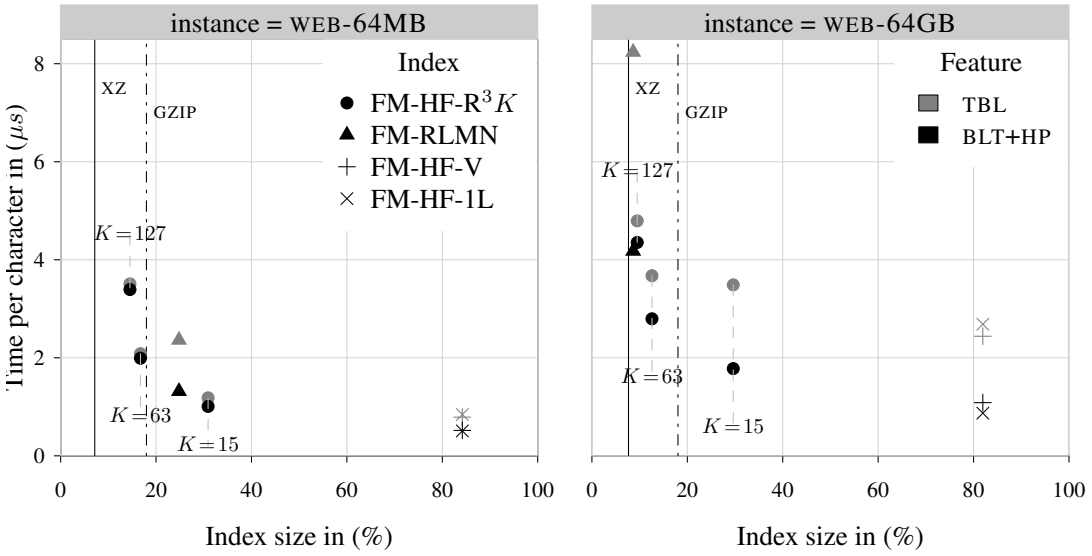

Figure 10. Count time and space of our index implementations on input instances of different size with compression effectiveness baselines using standard compression utilities XZ and GZIP with option `--best`.

FM-RLMN profits from each feature. Especially replacing the lookup table version for $popcnt$ and $select_{64}$ with more efficient BLT versions discussed above. Since FM-RLMN is a rather complex structure the different $rank$ and $select$ sub data structures "compete" for cache. As expected the FM-index based on RANK-1L is slower than FM-HF-V when no optimization is used but slightly faster when BLT and HP are activated. This reflects the outcome of the experiments of the basic $rank$ data structures used in both indexes shown in Figure 3.

Finally we discuss the time-space trade-offs of the different index types in Figure 10. As discussed above, optimizations BLT and hugepages only affects the runtime performance on larger data sets (right). The uncompressed representations FM-HF-V and FM-HF-1L, and FM-RLMN are affected the most by the features. The FM-indexes based on H_0 compressed bitvectors are also affected. Interestingly, the indexes for $K = 63$ and 128 achieve better compression then our `gzip --best` baseline. The FM-index using RANK-R^3K and $K = 128$ is only 5% larger than the `xz` compressed representation of the test input. The index for $K = 256$ is even smaller but is not shown in Figure 10 due to the increased runtime as shown in Table IX.

Overall our new data structures, improvements and optimized implementations enable FM-indexes that are faster than the state of the art (using RANK-1L) or smaller than state of the art indexes (using RANK-R^3K with $K = 128, 255$).

7. CONCLUSION

We proposed a simple, cache-friendly rank data structure, a practical select data structure for uncompressed bitvectors and an improved implementation of compressed bitvector representations. We explore the behavior of rank and select data structures for binary and general sequences for varying data sizes, implementations, instruction sets and operating system features. We show that using the built-in $popcnt$ and our optimized $select_{64}$ operation significantly improve the performance of classic rank and select data structures. We further discuss the effect of larger page sizes on the performance of succinct data structures. We demonstrate that these improvements propagate directly to more complex succinct data structures such as FM-indexes. Using larger page sizes has not yet been explored in literature in the context of succinct data structures and

DOI: 10.1002/spe

string processing. This is surprising, since succinct data structures tend to have memory access patterns which cause many TLB and cache misses. We think that exploiting the hugepage feature is an important step to making succinct structures competitive to classical uncompressed structures. The effects of using larger page sizes in the construction of index structures remains future work. However, first experiments show that the construction time of suffix array, Burrow-Wheeler Transform, longest common prefix array, and compressed suffix trees can be halved.

To foster the usage of succinct data structures we provide all our implementations in a ready-to-use open source C++ template library, sdsl.

ACKNOWLEDGEMENTS

We thank Zoltan Somogyi and Alistair Moffat for a fruitful discussions about virtual memory and Andrew Turpin for proof reading. We are also grateful to the anonymous reviewer who helped us discover the real state of the art implementation of select.

REFERENCES

1. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.
2. David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
3. Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. of the 15th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 176–187, 2008.
4. J. Shane Culpepper, Matthias Petri, and Falk Scholer. Efficient in-memory top-k document retrieval. In *Proc. of the 35th International ACM SIGIR conference on research and development in Information Retrieval (SIGIR)*, pages 225–234, 2012.
5. Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974.
6. Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
7. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.
8. Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly fm-index. In *Proc. of the 11th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 150–160, 2004.
9. Agner Fog. *Instruction tables*, 2012 (accessed March 13, 2012). `http://www.agner.org/optimize/instruction_tables.pdf`.
10. Simon Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, Ulm University, 2011.
11. Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. of 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 27–38, 2005.
12. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
13. Imran S. Haque, Vijay S. Pande, and W. Patrick Walters. Anatomy of high-performance 2d similarity calculations. *Journal of Chemical Information and Modeling*, 51(9):2345–2351, 2011.
14. Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compression, indexing, and retrieval for massive string data. In *Proc. of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 260–274, 2010.
15. Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988. AAI8918056.
16. Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in fm-indexes. In *Proc. of the 18th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011.
17. Donald Knuth. *The Art of Computer Programming, Volume 4A, The: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011.
18. Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 45–56, 2005.
19. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Proc. of the 13th International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 121–137, 2009.
20. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948, 1993.
21. Ian Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
22. Gonzalo Navarro. Wavelet trees for all. In *Proc. of the 23st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 2–26, 2012.
23. Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
24. Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proc. of the 11th International Symposium on Experimental Algorithms (SEA)*, pages 295–306, 2012.

DOI: 10.1002/spe

25. Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. of the 17th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.
26. Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of the 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2007.
27. Rasmus Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. Technical Report RS-98-28, BRICS, Department of Computer Science, University of Aarhus, 1998.
28. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.
29. Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
30. Alin Suciu, Petrut Cobarzan, and Kinga Marton. The never ending problem of counting bits efficiently. In *Proc. of the 10th Roedunet International Conference (RoEduNet)*, pages 1–4, 2011.
31. Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. of 7th Workshop on Experimental Algorithms (WEA)*, pages 154–168, 2008.

DOI: 10.1002/spe