Broadword Implementation of Rank/Select Queries

Sebastiano Vigna
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano, Italy

November 19, 2014

Abstract

Research on succinct data structures (data structures occupying space close to the information-theoretical lower bound, but achieving speed similar to their standard counterparts) has steadily increased in the last few years. However, many theoretical constructions providing asymptotically optimal bounds are unusable in practice because of the very large constants involved. The study of practical implementations of the basic building blocks of such data structures is thus fundamental to obtain practical applications. In this paper we argue that 64-bit and wider architectures are particularly suited to very efficient implementations of rank (counting the number of ones up to a given position) and select (finding the position of the i-th bit set), two essential building blocks of all succinct data structures. Contrarily to typical 32-bit approaches, involving precomputed tables, we use pervasively *broadword* (a.k.a. SWAR—"SIMD in A Register") programming, which compensates the constant burden associated to succinct structures by solving problems in parallel in a register. We provide an implementation named `rank9` that addresses 2^{64} bits, consumes less space and is significantly faster then current state-of-the-art 32-bit implementations, and a companion `select9` structure that selects in nearly constant time using only access to aligned data. For sparsely populated arrays, we provide a simple broadword implementation of the Elias–Fano representation of monotone sequences. In doing so, we develop broadword algorithms to perform selection in a word or in a sequence of words that are of independent interest.

1 Introduction

A *succinct* data structure (e.g., a succinct tree) provides the same (or a subset of the) operations of its standard counterpart, but occupies space that is asymptotically near to the information-theoretical lower bound. A classical example is the $(2n + 1)$-bit representation of a binary tree with n internal nodes proposed by Jacobson [Jac89]. Recent years have witnessed a growing interest in succinct data structures, mainly because of the explosive growth of information in various types of text indexes (e.g., large XML trees).

In this paper we discuss practical implementations of two basic building blocks—*rank* and *select*. Given an array B of n bits, we are interesting in *ranking* the i-th position (computing the number of ones up to that position) and *selecting* the i-th bit set to one.

It is known that with an auxiliary data structure occupying $o(n)$ bits it is possible to answer both rank and select queries in constant time (see, e.g., [Gol06] and references therein for an up-to-date overview). A complementary approach discards the bit vector altogether, and stores explicitly the positions of all ones in a *fully indexable dictionary*, which represents a set of integers making it possible to access the k-th element of the set in increasing order, and to compute the number of elements of the set smaller than a given integer. These two operations correspond to selection and ranking over the original bit vector: by using succinct dictionaries, it is possible to reduce significantly the space occupancy with respect to an explicit bit vector in the sparse case.

We start from concerns similar to those of González, Grabowski, Mäkinen and Navarro [GGMN05]: it is unclear whether these solutions are usable in practice. The asymptotic notation is often hiding constants so large that before the asymptotic advantage actually kicks in, the data structure is too large. In this case, it is rather fair to say that the result is interesting mathematically, but has little value as a data structure.

This problem is made even worse by the fact that succinct data structure are exactly designed for very large data sets, which are useless if the access to the data is slow. For instance, the authors of [GGMN05] argue that word-aligned, $O(n)$ solutions are extremely more efficient than the optimal counterparts, and that for perfectly reasonable data sizes they actually occupy less space. To solve locally (wordwise) rank and select the author use *population counting* techniques—precomputed tables containing, say, the number of bits set to one in each possible byte.

In this paper we depart from this approach, arguing that on modern 64-bit architecture a much more efficient approach uses *broadword programming*. The term "broadword" has been introduced by Don Knuth in the fascicle on bitwise manipulation techniques of the fourth volume of *The Art of Computer Programming* [Knu07]. Broadword programming uses large (say, more than 64-bit wide) registers as small parallel computers, processing several pieces of information at a time. An alternative, more traditional name for similar techniques is SWAR ("SIMD Within A Register"), a term coined by Fisher and Dietz [FD99]. One of the first techniques for manipulating several bytes in parallel were actually proposed by Lamport [Lam75].

For instance, a broadword algorithm for *sideways addition* (counting the number of ones in a register—of course, part of computing ranks) was presented in the second edition of the textbook "Preparation of Programs for an Electronic Digital Computer", by Wilkes, Wheeler, and Gill, in 1957. One of the contributions of this paper is a broadword counterpart to select bits in a word.

The main advantage of broadword programming is that we gain more speed as word width increases, with almost no effort, because we can process more data in parallel. Note that, in fact, broadword programming can even be used to obtain better asymptotic results: it was a basic ingredient for the success of *fusion trees* in breaking the information-theoretical lower bound for integer sorting [FW93].

Using broadword programming, we are able to fulfil at the same time the following apparently contradictory goals:

- address 2^{64} bits[1]
- use less space;
- obtain faster implementations.

A second concern we share with the authors of [GGMN05] is that of minimising cache misses, as memory access and addressing is the major real bottleneck in the implementation of rank/select queries on large-size arrays. To that purpose, we *interleave* data from different tables so that usually a single cache miss is sufficient to find all information related to a portion of the bit array (we wish to thank one of the anonymous referees for pointing out that the idea already appeared in [GRRR06]).

We are also very careful of avoiding tests whenever possible. Branching is a very expensive operation that disrupts speculative execution, and should be avoided when possible. All the broadword algorithms we discuss contain no test and no branching.

We concentrate on 64-bit and wider architecture, but we cast all our algorithms in a 64-bit framework to avoid excessive notation: the modification for wider registers are trivial. We have in mind modern processors (in particular, the very common Opteron processor) in which multiplications are extremely fast (actually, because the clock is slowed down in favour of multicores), so we try to use them sparingly, but we allow them as constant-time operations. While this assumption is debatable on a theoretical ground, it is certainly justified in practice, as experiments show that on the Opteron replacing multiplications by shifts and additions, *even in very small number*, is not competitive.

The C++/Java code implementing all data structures in this paper is available under the terms of the GNU Lesser General Public License at `http://sux.di.unimi.it/`.

[1] All published practical implementations we are aware of address 2^{32} bits; this is a serious limitation, in particular for compressed structures.

2 Notation

Consider an array $\boldsymbol{b}$ of n bits numbered from 0. We write b_i for the bit of index i, and define

$$\operatorname{rank}_{\boldsymbol{b}}(p) = \sum_{0 \le i < p} b_i \qquad 0 \le p \le n,$$

that is, as the number of ones up to position p, excluded, and

$$\operatorname{select}_{\boldsymbol{b}}(r) = \max\{\, p < n \mid \operatorname{rank}_{\boldsymbol{b}}(p) \le r \,\}, \qquad 0 \le r < \operatorname{rank}_{\boldsymbol{b}}(n),$$

that is, as the position of the one of index r, where ones are numbered starting from 0. When $\boldsymbol{b}$ is clear from the context, we shall omit it.

Note that in the literature there is some variation in the choice of indexing (starting from one or zero) and in the exact definition of these two primitives (including or not the one at position p in $\operatorname{rank}(p)$).

To be true, we couldn't find the 0-based definitions given above in the literature, but they are extremely natural for several reasons:

- As it always happen with modular arithmetic, starting with 0 avoids falling into "off-by-one hells". This consideration is of course irrelevant for a theoretical paper, but we are in a different mindset.
- In this way, $\operatorname{rank}(p)$ can be interpreted as $\operatorname{rank}[0\,..\,p)$—counting the ones in the semiopen interval $[0\,..\,p)$. Counting from zero and semiopen intervals are extremely natural in programming (actually, Dijkstra felt the need to write a note on the subject [Dij82]).
- We can define easily, and without off-by-ones, operators such as $\operatorname{count}_{\boldsymbol{b}}[p\,..\,q) = \operatorname{rank}(q) - \operatorname{rank}(p)$.

In any case, it is trivial to compute other variations of rank and select by suitably offseting the arguments and the results.

We use $a \setminus b$ to denote integer division of a by b, $\gg$ and $\ll$ to denote right and left (zero-filled) shifting, &, | and $\oplus$ to denote bit-by-bit not, and, or, and xor; $\overline{x}$ denotes the bit-by-bit complement of x. We pervasively use precedence to avoid excessive parentheses, and we use the same precedence conventions of the C programming language: arithmetic operators come first, ordered in the standard way, followed by shifts, followed by logical operators; $\oplus$ sits between | and &.

We use L_k to denote the constant whose ones are in position 0, k, $2k$, … that is, the constant with the *lowest* bit of each k-bit subword set (e.g, $L_8 = 0x0101010101010101$). This constant is very useful both to spread values (e.g., $12 * L_8 = \mathrm{0x1212121212121212}$) and to sum them up, as it generates cumulative sums of k-bit subwords if the values contained in each k-bit subword, when added, do not exceed k bits. (e.g., $\mathrm{0x030702} * L_8 = \mathrm{0x30A0C0C0C0C0C0902}$—look carefully at the three rightmost bytes). We use H_k to denote $L_k \ll k - 1$, that is, the constant with the *highest* bit of each k-bit subword set (e.g, $H_8 = 0x8080808080808080$).

Our model is a RAM machine with d-bit words that performs logic operations, additions, subtractions and multiplications in unit time using 2-complement arithmetic. We note that albeit multiplication can be proven to require $O(\log d)$ basic operations, modern processors have very fast multiplication (close to one cycle), so designing broadword algorithms without multiplications turns out to generate slower code.

3 `rank9`

We now introduce the layout of our data structure for ranking, which follows a traditional two-level approach but uses broadword sideways addition (Algorithm 1) for counting inside a word and interleaving to reduce cache misses. We assume the bit array $\boldsymbol{b}$ is represented as an array of words of 64 bits. The bit of position p is located in the word of index $p \setminus 64$ at position $p \bmod 64$, and we number bits inside each word in *little-endian* style.

To each subsequence of eight words starting at bit position p, called a *basic block*, we associate two words:

- the first word (first-level count) contains $\mathrm{rank}(p)$;
- the second contains the seven 9-bit values (second-level counts) $\mathrm{rank}(p + 64k) - \mathrm{rank}(p)$, for $1 \le k \le 7$, each shifted left by $9(k - 1)$ bits.

First and second level counts are stored in *interleaved form*—each first-level count is followed by its second-level counts. When we have to rank a position p living in the word $w = p \setminus 64$, we have just to sum the first-level count of the sequence starting at $w \setminus 8$, possibly a second-level count (if $w \bmod 8 \neq 0$) and finally invoke sideways addition on the word containing p, suitably masked. Note that this apparently involves a test, but we can get around the problem as follow:

$$s \gg (t + (t \gg 60 \mathbin{\&} 8)) * 9 \mathbin{\&} \mathrm{0x1FF},$$

where s is the second-level count and $t = w \bmod 8 - 1$. When $w \bmod 8 = 0$, the expression $t \gg 60 \mathbin{\&} 8$ has value 8, which implies that s is shifted by 63, obtaining zero (we are not using the most significant bit of s).

We call the resulting structure `rank9` (the name, of course, is inspired by the fact that it stores 9-bit second-level counts). It requires just 25% additional space, and ranks are evaluated with at most *two* cache misses, as when the first-level count is loaded by the L1 cache, the second-level count is, too. No tests or precomputed tables are involved.[2]

The only dependence on the word length d is in the first cumulative phase of sideways addition. We need to cumulate at least b bits, where b is a power of two enough large to express d, that is, $b = \lceil \log d \rceil$. Thus, this phase requires $O(\log \log d)$ step. However, since Algorithm 1, with suitable constants, works up to $d = 256$, it can be considered constant time to all practical purposes (as we will never have 2^{256} bits).

Algorithm 1 The classical broadword algorithm for computing the sideways addition of x in $O(\log \log d)$ steps. The first step leaves in each pair of bits the number of ones originally contained in that pair. The following steps gather partial summations and, finally, the multiplication sums up them all.

$x = x - ((x \mathbin{\&} \mathrm{0xAAAAAAAAAAAAAAAA}) \gg 1)$
$x = (x \mathbin{\&} \mathrm{0x3333333333333333}) + ((x \gg 2) \mathbin{\&} \mathrm{0x3333333333333333})$
$x = (x + (x \gg 4)) \mathbin{\&} \mathrm{0x0F0F0F0F0F0F0F0F}$
$x * L_8 \gg 56$

4 Interlude: k-bit comparisons

Given x and y, consider them as sequences of $64 \setminus k$ (un)signed k-bit values. We would like to operate on them so that, in the end, each k-bit block contains 1 in the leftmost position iff the corresponding pair of k-bit values is ordered. At that point, it is easy to count how many ones are present using a multiplication. Knuth describes a broadword expression to this purpose, using the properties of the median (a.k.a. majority) ternary operator [Knu07]. Here, to make the paper self-contained we

[2] Of course, if more than 64 bits per word are available, more savings are possible: for instance, for 128-bit processors `rank16`, which uses 16-bit second-level counts, requires just 12.6% additional space.

provide an elementary independent construction which uses the same number of operators, but has the (small) advantage that no subexpression appears twice except for positive literals. This makes our formulation ideal for macro expansion in C.

We recall the expression for computing in parallel the differences modulo 2^k of each k-bit subword:

$$((x \mid H_k) - (y \,\&\, \overline{H_k})) \oplus ((x \oplus \overline{y}) \,\&\, H_k).$$

If all k-bit subwords contain values smaller than 2^{k-1}, the most significant bit of each subword will be set to one iff the corresponding subwords of x and y are strictly ordered. After anding with H_k and some simple manipulation we have:

$$\begin{aligned}
&\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \oplus ((x \oplus \overline{y}) \,\&\, H_k)\big) \,\&\, H_k \\
&\qquad = ((x \mid H_k) - (y \,\&\, \overline{H_k}) \,\&\, H_k) \oplus ((x \oplus \overline{y}) \,\&\, H_k) \\
&\qquad\qquad = \big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \oplus x \oplus \overline{y}\big) \,\&\, H_k,
\end{aligned}$$

so we define

$$x <_k y := \big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \oplus x \oplus \overline{y}\big) \,\&\, H_k.$$

It is immediate to check that $x <_k y$ *does* work if either $x, y < 2^{k-1}$ or $x, y \geq 2^{k-1}$. Thus, we are just left to fix manually the remaining cases: if $x < 2^{k-1}$ and $y \geq 2^{k-1}$, however, we already known the result, so oring the value $((\overline{x} \,\&\, H_k) \,\&\, (y \,\&\, H_k)) \,\&\, H_k = \overline{x} \,\&\, y \,\&\, H_k$ would fix that case; analogously, anding it with $\overline{((x \,\&\, H_k) \,\&\, (\overline{y} \,\&\, H_k))} \,\&\, H_k = (\overline{x} \mid y) \,\&\, H_k$ would fix the dual case. Since in the end the bits in H_k are all that matters, we are interested in computing

$$\begin{aligned}
&(x <_k y) \,\&\, (\overline{x} \mid y) \,\&\, H_k \mid (\overline{x} \,\&\, y \,\&\, H_k) \\
&\qquad = \Big(\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \oplus x \oplus \overline{y}\big) \,\&\, (\overline{x} \mid y) \mid (\overline{x} \,\&\, y)\Big) \,\&\, H_k \\
&\qquad\qquad = \Big(\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \,\&\, (\overline{x} \mid y)\big) \oplus \overline{x \oplus y} \mid (\overline{x} \,\&\, y)\Big) \,\&\, H_k,
\end{aligned}$$

where the last passage follows by distributing & over $\oplus$ and reducing the resulting terms. To reduce further, we use De Morgan to turn | into &; so since $(\overline{x} \mid y) \,\&\, (x \mid \overline{y}) = \overline{x \oplus y}$ we have

$$\begin{aligned}
&\overline{\Big(\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \,\&\, (\overline{x} \mid y)\big) \oplus x \oplus y\Big)} \,\&\, (x \mid \overline{y}) \,\&\, H_k \\
&\qquad = \overline{\Big(\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \,\&\, \overline{x \oplus y}\big) \oplus ((x \oplus y) \,\&\, (x \mid \overline{y}))} \,\&\, H_k \\
&\qquad\qquad = \Big(\big(\overline{((x \mid H_k) - (y \,\&\, \overline{H_k}))} \mid x \oplus y\big) \oplus x \,\&\, \overline{y}\Big) \,\&\, H_k.
\end{aligned}$$

We now notice that if we replace & with | in the term $x \,\&\, \overline{y}$, we flip the value of the term exactly when $x = y$. However, in that case the value of the remaining part is just depending on $\overline{((x \mid H_k) - (y \,\&\, \overline{H_k}))}$, so we can compensate the flip by removing the outermost negation (when $x \neq y$ this removal has no effect). We thus define

$$x <_k^u y := \Big(\big(((x \mid H_k) - (y \,\&\, \overline{H_k})) \mid x \oplus y\big) \oplus (x \mid \overline{y})\Big) \,\&\, H_k.$$

This eight-operator formula has the nice property that no subexpression (except for positive literals and constants) appears twice, so it can be used for macro expansion without caching intermediate values. Exhaustive search shows that, given x, y and $(x \mid H_k) - (y \,\&\, \overline{H_k})$, this eight-operator formula is the shortest possible. If we need $\leq_k^u$, we just interchange the rôles of x and y and complement the result:

$$\begin{aligned}
x \leq_k^u y &:= \Big(\overline{\big(((y \mid H_k) - (x \,\&\, \overline{H_k})) \mid y \oplus x\big) \oplus (y \mid \overline{x})}\Big) \,\&\, H_k \\
&\qquad = \Big(\big(((y \mid H_k) - (x \,\&\, \overline{H_k})) \mid x \oplus y\big) \oplus (x \,\&\, \overline{y})\Big) \,\&\, H_k.
\end{aligned}$$

Analogously,

$$x \le_k y := \left(\overline{((y \mid H_k) - (x \mathbin{\&} \overline{H_k})) \oplus y \oplus \overline{x}}\right) \mathbin{\&} H_k$$
$$= \left(((y \mid H_k) - (x \mathbin{\&} \overline{H_k})) \oplus x \oplus y\right) \mathbin{\&} H_k.$$

In the particular case $x = 0$ (i.e., we want to know which k-bit blocks contain nonzero data), $x >^u_k 0$ simplifies considerably, getting to a four-operator formula (analogously to what happens in [Knu07]), that we obtain easily as a by-product; actually, is much easier to simplify $x \ge^u_k L_k$:

$$x >^u_k 0 = x \ge^u_k L_k = \Big(\big(((x \mid H_k) - (L_k \mathbin{\&} \overline{H_k})) \mid L_k \oplus x\big) \oplus (L_k \mathbin{\&} \overline{x})\Big) \mathbin{\&} H_k$$
$$= \Big(\big(((x \mid H_k) - L_k) \mid L_k \oplus x\big) \oplus (L_k \mathbin{\&} \overline{x})\Big) \mathbin{\&} H_k$$
$$= \Big(\big(((x \mid H_k) - L_k) \mid x\big)\Big) \mathbin{\&} H_k.$$

5 `select9`

We would like to build upon `rank9` selection capabilities. To this purpose, we work backwards, starting from selection in a word, moving to selection in a sequence of words, and finally getting to selection over the bit array. In `rank9` we conceded a shift-based access to non-aligned subwords, but in the case of select several accesses are needed (even in the optimal, non-aligned data structures), so we will limit ourselves to access only correctly aligned subwords of size $d/2^i$ (except, of course, for `rank9` access).

The starting consideration for our select-in-a-word broadword algorithm is the observation that at the end of Algorithm 1 we use just the most significant byte of a multiplication that provides much more information—namely, the cumulative sums of the number of ones contained in each byte. If we compare each of these numbers with the desired index r, we can easily locate the byte containing the r-th one. With a typical broadword approach, we then solve the problem in the relevant byte in a similar manner.

We are now ready to introduce Algorithm 2. In the first lines we follow exactly Algorithm 1, building the bytewise cumulative sums s. Then, we compare in parallel each cumulative sum with r: the number of positive results is exactly the index of the byte containing the bit of rank r, so we extract it in b already multiplied by eight. To obtain the bytewise rank ℓ, we subtract from r the value found in the byte starting at bit $b - 8$ (if $b = 0$, $\ell = r$).

We now compute a word z that contains eight copies of the byte starting at position b (the one containing the bit of rank r); however, from the j-th copy we just keep bit j. We now compare each byte in parallel with zero, which make it possible to compute, with a multiplication by L_8, the rank of each bit. We compare the cumulative sums with eight copies of ℓ; again, the number of positive results is the index of the ℓ-th one, which we return, summed with b.

We note that, similarly to sideways addition, we need to compute the number of ones in subwords of size $\lceil \log d \rceil$. Now, however, we have another constraint: $\lceil \log d \rceil$ copies of each sum must fit into a word, that is, $\lceil \log d \rceil^2 \le d$. This constraint cannot be satisfied with d a power of two unless $d \ge 64$.

Again, Algorithm 2 requires $O(\log \log d)$ operations in the initial phase, and up to $d = 256$ the only modifications required are suitable changes to the constants. Moreover, the constant operations significantly outnumber those of the initial phase. Finally, the algorithm contain several multiplications by L_8: they can be replaced by less than $\log d$ shifts and adds, as the number of ones in L_8 is very low.

We now approach the problem of constant-time selection inside a block of `rank9`. The idea, by now familiar to the reader, is to locate the right word using parallel comparisons. More precisely, if s contains the subcount word and we have to locate the bit of rank r we can just compute

$$o = ((s \le^u_9 r * L_9) \gg 8) * L_9 \gg 54 \mathbin{\&} 7$$

Algorithm 2 Computes the index of the r-th one in x ($r < 2^{2^{\lceil \log \log d \rceil}}$). If no such bit exists, computes 72.

$$s = x - ((x \mathbin{\&} \text{0xAAAAAAAAAAAAAAAA}) \gg 1)$$
$$s = (s \mathbin{\&} \text{0x3333333333333333}) + ((x \gg 2) \mathbin{\&} \text{0x3333333333333333})$$
$$s = ((s + (s \gg 4)) \mathbin{\&} \text{0x0F0F0F0F0F0F0F0F}) * L_8$$
$$b = ((s \leq_8 r * L_8) \gg 7) * L_8 \gg 53 \mathbin{\&} \overline{7}$$
$$\ell = r - (((s \ll 8) \gg b) \mathbin{\&} \text{0xFF})$$
$$s = (((x \gg b \mathbin{\&} \text{0xFF}) * L_8 \mathbin{\&} \text{0x8040201008040201} >_8 0) \gg 7) * L_8$$
$$b + (((s \leq_8 \ell * L_8) \gg 7) * L_8 \gg 56)$$

to know the offset in the block of the word containing the bit, and

$$r - (s \gg (o - 1 \mathbin{\&} 7) * 9 \mathbin{\&} \text{0x1FF})$$

to know the rank inside the word. Note that $o - 1 \mathbin{\&} 7$ is 63 when $o = 0$, which implies that no correction is performed if the bit belongs to the first word in the block.

Binary-search selection. At this point, we could follow the steps of [GGMN05] and just perform a binary search over blocks, followed by the broadword block search we just described. Moreover, we could add a simple, one-level inventory that would help locating more quickly the region in which perform a binary search: we call this approach a *hinted bsearch*. In the experimental part, however, we will see that while (hinted) binary searches have excellent performances on evenly distributed arrays, they give worst results on uneven distributions.

Selecting in $d\sqrt{d}$ words. In general, the approach we described provides selection in $\sqrt{d}$ words. We are now going to use the broadword approach to provide selection in practical constant time inside $d\sqrt{d}$ words.

The idea is very simple: since by broadword comparison we can quickly locate, in a list of increasing integers, the first integer larger than a given integer x, given a sequence of $\sqrt{d}$ basic blocks, that we shall call an *intermediate block*, we can list the $\sqrt{d}$ first-level count of each block and perform selection by first locating the correct basic block, and then operating as we previously described. Note that since we need just to store the difference of each first-level count from the first one, we need very few bits ($2 \log d$), so a constant number of words will suffice. In our main example, we use two words to store eight 16-bit values containing the first-level counts.

To get to $d\sqrt{d}$ words (512, in our example) we repeat again the same trick, but now we consider a sequence of $\sqrt{d}$ intermediate blocks, called an *upper block*, and record the $\sqrt{d}$ first-level counts of the first basic block of each intermediate block. Using the parallel comparison operator as we did in the first part of this section, and using suitable constants (e.g., L_{16}) we can find in constant time the intermediate block and, again in constant time, the basic block containing the bit we are interested in.

We note that the cost of recording this information is very low: when $d = 64$ we need 16 bits for each basic block, which contains 512 bits.

Selecting over the whole bit array. Our interest in selecting over $d\sqrt{d}$ words stems from the fact that, by keeping track of the position of one each $d\sqrt{d}$ bits in a primary inventory space and allocating with care some secondary inventory, we can reduce in constant time our problem to selection in $d\sqrt{d}$ words.

More precisely, we record the position of each $d\sqrt{d}$-th bit. In our example, in the worst case (density close to 1) this information requires 12.5% additional space. Then, we allocate one word each α words for a secondary inventory. Consider two bits that appear consecutively in the primary inventory (in particular, their indices differ by $d\sqrt{d}$), and let p and q be their positions. For the $d\sqrt{d}$ bits inbetween we have at our disposal

$$q \backslash (\alpha d) - p \backslash (\alpha d)$$

words. If this number is at least $d\sqrt{d}$, we can record the position of each bit. Otherwise, we can describe the position of each bit in this range using

$$\log(\alpha d^2\sqrt{d}) = \log\alpha + \frac{5}{2}\log d$$

bits, so as long as

$$\log(\alpha d^2\sqrt{d}) = \log\alpha + \frac{5}{2}\log d \leq \frac{d}{2}$$

we can still describe the position of each bit using the upper and lower half of each word (note that, as we discussed, we are purposely avoiding to manipulate non-aligned subwords). The process can continue if there is enough space to describe the position of all $d\sqrt{d}$ bits: depending on α, more or less subword sizes can be used.

For the case $d = 64$, $\alpha = 4$ is a particularly good value because it generates an equality in the inequality

$$\log a + \frac{5}{2}\log d - 1 \leq \frac{d}{4},$$

which means that we can get to the point where we are recording the positions of all $d\sqrt{d} = 512$ bits using 128 words of secondary storage. Since these 128 words correspond to $512 = d\sqrt{d}$ words in our bit array, below this size we can use the broadword techniques described in the previous paragraph.

All in all, `select9` uses an underlying `rank9` structure, plus additional data occupying at most 37.5% of the original bit array. To rank a bit r, we first compute the positions p and q of the bit $r' = r - (r \bmod d\sqrt{d})$ and of the bit $r' + d\sqrt{d}$, respectively, using the primary inventory. Then, we compute the *span* associated to r'

$$s = (q\backslash d)\backslash\alpha - (p\backslash d)\backslash\alpha,$$

which represent the number of words from the secondary inventory we can use for the $d\sqrt{d}$ bits after r'. Finally, to locate the position of the bit of position r, we proceed as follows:

1. if $s < 2$, the bit can be located inside the basic block to which r' belongs;
2. if $s < 16$, the bit can be located using a two-word index collecting the first-level counts of an intermediate block;
3. if $s < 128$, the bit can be located using an eighteen-word two-level index collecting the first-level counts of an upper block, organised as we described above; note that by storing the two indices consecutively, we effectively interleave the data, generating a single cache miss for both reads;
4. if $s < 256$, we store explicitly the offset of each bit from r' (whose rank is known by first-level counting) in 16 bits;
5. if $s < 512$, we store explicitly the offset of each bit in 32 bits;
6. otherwise, we have enough space to store explicitly all bit positions.

It is easy to check that the choice $\alpha = 4$ makes it possible to store any of the alternative information required by the data structure.

In the worst case, `select9` will generate four cache misses: one to access the primary inventory, one to access the secondary inventory, one to locate the correct basic block, and one to select inside a basic block. The only test required when performing selection is comparing the value of s with the constants above.[3]

[3] We remark that we claimed in the introduction that our broadword algorithms contain no branching; but there is no contradiction, as this part of `select9` is not broadword.

6 `simple`

The idea of broadword selection can be easily extended to a *bit search* algorithm that quickly locates a bit in a bit array. Assuming we want to locate the bit of rank r in a sequence of words, we simply have to load the first word into x and loop around the first three lines of Algorithm 2: if $r < s \gg 56$, we exit the loop and proceed as usual. Otherwise, we load x with the content of the next word, decrease r by $s \gg 56$ and iterate again.

Armed with this tool, we implement `simple`, an almost naive but surprisingly efficient select structure that does not depend on `rank9`. The structure is a two-level inventory similar to the `darray` dense select structure described in [OS07], but it has been suitably modified to have reduced access time an halved space occupancy in spite of 64-bit addressing.

We keep an inventory of ones at position multiples of $\lceil Lm/n \rceil$, where L is a constant limiting the size of the inventory ($L = 8192$ in our implementation). For each bit in the inventory, we allocate a number of words (again, upper bounded by a constant M) depending on the density. Inside, we record a 16-bit subinventory (if 16 bits are not enough, we use the space to point at a spill buffer where we record each bit position individually). We use the inventory and the subinventory to locate a position that is near the bit we intend to select, and then we perform a linear broadword bit search. The experimental results about this algorithm show that, in fact, it is the fastest, even in the presence of uneven bit distribution. It also has the advantage of providing just selection with a very limited space usage.

The memory occupancy depend mainly by the bound M. Due to the speed of broadword bit search, we have been able to halve it with respect to the value used in [OS07], without a noticeable effect on performance. As a result, we have almost halved the space occupancy.

7 Elias–Fano representation of monotone sequences

For sparse arrays, we provide a 64-bit implementation of the Elias–Fano representation of monotone sequences [Eli74, Fan71], which is one of the earliest examples of a fully indexable dictionary. We briefly recall the main idea, translated into the bit array scenario: we record all bits positions, but while the lower $\ell = \lfloor \log(n/m) \rfloor$ bits are recorded explicitly, the $u = \lceil \log n \rceil - \lfloor \log(n/m) \rfloor$ upper bits are recorded in an array U of $m + u \setminus 2^\ell$ bits as follows: if the value of the upper u bits of the position of the i-th one is k, we set the bit in position $i + k$. It is easy to recover the original value by selecting the i-th bit in U and subtracting i. The space occupancy is bounded by $2m + m\log(n/m)$ bits [Eli74], which is almost optimal as specifying a set of m elements out of n requires $\approx m\log(n/m)$ bits when $m \ll n$.[4]

The only component we can improve is actually selection in U, which however is a very well-behaved dense array, so we use a version of `simple` that is wired to density 1/2.

8 Experiments

We performed a number of experiments on a Linux-based system sporting a 64-bit Opteron processor running at 2814.501 MHz with 1 MiB of first-level cache. The tests show that on 64-bit architectures broadword programming provides significant performance improvements. We compiled using `gcc` 4.1.2 and options `-O9`.

The experimental setting for benchmarking operations that require few nanoseconds must be set up carefully. We generate random bit arrays and store a million test positions. During the tests, the positions are read with a linear scan, producing a minimal interference; generating random positions during the tests causes instead a significant perturbation of the results, mainly due to the slowness of

[4]Actually, we are somewhat wasting space, as the Elias–Fano representation can code sequences with duplicates, and in that case the lower bound is $\approx m\log e + m\log(n/m)$. As already suggested by Elias [Eli72], it is easy to map bijectively a strictly increasing sequence of m elements upper bounded by n into a nondecreasing sequence of m elements upper bounded by $n - m$ and apply the representation to the latter sequence, but that would make ranking more difficult.

the modulo operator. The tests are repeated ten times and averaged. We measure user time using the system function `getrusage()`.

We provide results for dense (50%) and sparse (1%) arrays of different sizes[5]. In the first case, however, we take care of experimenting over a highly uneven bit array (almost empty in the first half, almost full in the second half). Test positions are generated so to fall approximately half of the time in the dense part, and half of the time in the sparse part. The results obtained using this method highlight serious limitations of some approaches (e.g., binary search) which are not evident in experiments involving uniform bit arrays. Our results suggest that practical implementations of rank/select queries should be always tested against uneven bit arrays (and possibly even more adversarial settings).

We chose to compare our structures against practical ones: the code for the BitRankF structure proposed in [GGMN05] was provided by the authors. The authors of [OS07] provided code for their implementation of the Elias–Fano[6] representation (`darray`[7] and `sarray`), and for the byte-oriented select structure described by Kim *et al.* in [KNKP05].[8] All these structures exploit byte or word alignment to increase speed, as previous experiments have made clear [GGMN05] that non-aligned structures are extremely slow. Nonetheless, to let the reader have a feeling about what happens using $o(n)$-space constant-time structures we also provide results about Jacobson's [Jac89] classic rank implementation and Clark's [Cla98] select implementation.[9][10]

Looking at Table 2, `rank9` is the clear winner among ranking methods. For completeness, we provide results for a variant that trades broadword programming for *population counting* ("pc"), a standard table-based technique used in [GGMN05] that turns out to be slower.[11] The situation for select is more varied, and also Table 3 and 4 should be taken into account. Essentially, `simple` turns out to be the fastest and more space efficient data structure on evenly distributed arrays. If constant time is required in spite of adversarial distribution, `select9` is highly competitive if paired with `rank9`.

The results for selection on sparse arrays are reported in Table 5 and 6. Our implementation of the Elias–Fano representation provides support for very large (64-bit) arrays while keeping the excellent space occupancy of `sarray` (for lack of space we cannot report results on ranking, which are however in the same line). Among implementations requiring the original bit array, `select9` has excellent performance even on very large arrays. Its space occupancy is also very competitive if it used in conjunction with `rank9`, albeit `simple` has also very good timings, and the lowest space occupancy.

9 Conclusions

We have introduced some new ideas about the application of broadword programming [Knu07] to bit-level manipulations typical of succinct static data structures. We have also presented exhaustive

[5]Note that we use the NIST-endorsed prefixes: Ki=2^{10}, Mi=2^{20}, etc.

[6]It should be noted that in [OS07] no mention is made of the work of Elias and Fano. Moreover, their bit subdivision (using $\lceil \log(n/m) \rceil$ lower bits) causes a larger space occupation.

[7]We have decreased the bound M in `darray` to reduce further space occupancy; we can do so with an almost immaterial impact on performance due to the speed of broadword bit search.

[8]The authors of the latter paper, in spite of several communication attempts, did not provide code for their structures.

[9]The code for the latter was kindly provided by the authors of [GGMN05].

[10]We wish to thank one of the anonymous referees for pointing us at a series of papers about practical rank/select structures [GRRR06, DRR07]. Unfortunately, at the time of this writing the authors distribute publicly just a few header files and two binary libraries for an unspecified operating system, without any source code or documentation.

[11]It is interesting to remark that testing in isolation broadword programming *vs.* popcounting for ranking or selecting in a word we obtained opposite results. This happens because when testing popcounting in isolation the whole processor cache and branch-prediction unit are servicing a single, small loop.

Size	`select9`	Hinted bsearch	`simple`	`darray`	Kim	Clark
1 Ki	62.50%	37.50%	25.00%	67.19%	86.72%	544073.24%
16 Ki	56.25%	37.50%	14.45%	28.81%	72.80%	34074.85%
256 Ki	56.13%	37.23%	13.79%	27.73%	71.57%	2184.99%
4 Mi	56.12%	37.25%	13.78%	27.56%	71.67%	192.81%
64 Mi	56.12%	37.25%	13.78%	27.56%	71.67%	68.30%
1 Gi	56.13%	37.25%	13.78%	27.56%	71.67%	60.52%

Table 1: Percentage of space occupied by various select structures in a densely (50%) populated bit array. Note that the percentage shown for `select9` and hinted bsearch includes 25% for `rank9`. The preposterous values shown for Clark's structure are due to the very large lookup table.

experimentation that compares our results with approaches existing in the literature, providing also some new, effective ideas for testing. Beside achieving 64-bit size, we can at the same time significantly improve (sometimes both) speed and space occupancy.

For densely populated arrays, `rank9` and `simple` are generally the best structures, both in term of time, space, and addressability. If a more robust performance guarantee is required, `select9` provide the fastest practical constant-time operations. For sparsely populated arrays, the Elias–Fano representation of monotone sequences, supported by dense broadword selection, provides good speed and nearly optimal space occupancy.

We wish to thank one of the anonymous referees for pointing us to Elias's paper [Eli74], which in turn led us to Fano's *memorandum* [Fan71].

References

[Cla98] David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, 1998.

[Dij82] Edsger W. Dijkstra. Why numbering should start at zero, 1982. EWD 831.

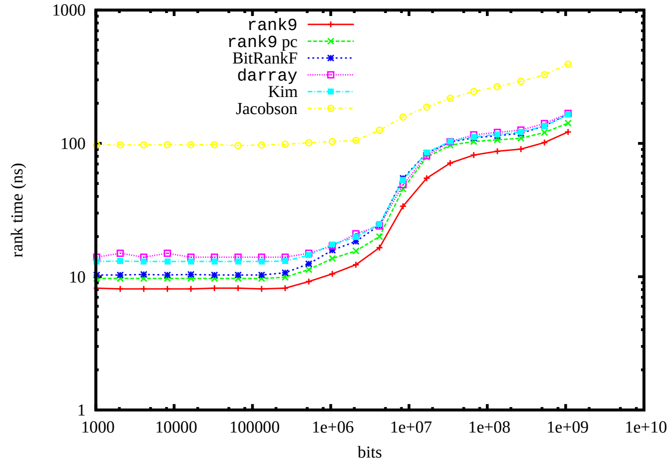
[DRR07] O'Neil Delpratt, Naila Rahman, and Rajeev Raman. Compressed prefix sums. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *Proc. SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science*, number 4362 in Lecture Notes in Computer Science, pages 235–247. Springer–Verlag, 2007.

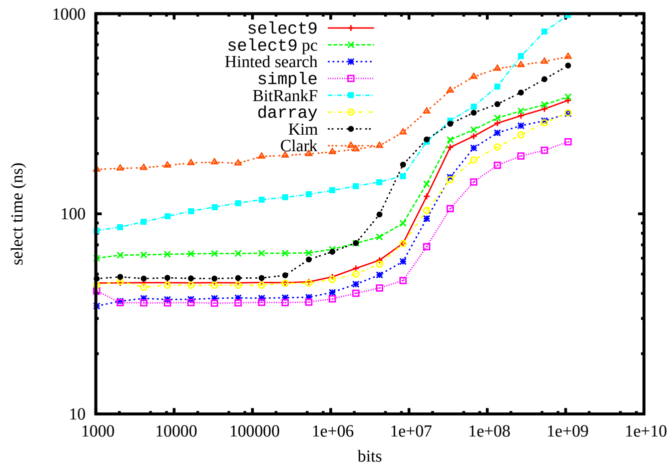
[Eli72] Peter Elias. On binary representations of monotone sequences. In *Proc. Sixth Princeton Conference on Information Sciences and Systems*, pages 54–57, Dep. of Electrical Engineering, Princeton U., Princeton, N. J., 1972.

[Eli74] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.*, 21(2):246–260, 1974.

[Fan71] Robert M. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d., 1971.

[FD99] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD within a register. In Siddhartha Chatterjee, Jan Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David C. Sehr, and Pen-Chung Yew, editors, *Languages and Compilers for Parallel Computing, (11th LCPC'98)*, number 1656 in Lecture Notes in Computer Science, pages 290–304. Springer–Verlag, 1999.

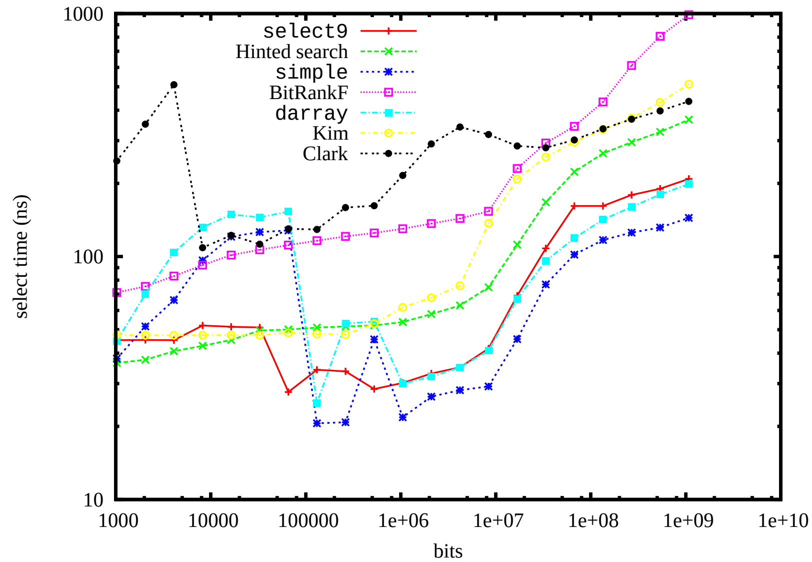




Size	`rank9` (25%)	`rank9` pc	BitRankF (37.5%)	Kim (37.5%)	`darray` (25%)	Jac. (> 66.85%)
1 Ki	8.2	9.7	10.3	14.0	13.1	97.8
16 Ki	8.1	9.8	10.3	14.0	13.0	97.6
256 Ki	8.3	9.9	10.6	14.0	13.3	98.5
4 Mi	16.5	19.4	25.7	25.0	24.3	123.5
64 Mi	81.9	103.7	110.3	115.0	112.8	245.1
1 Gi	121.1	141.1	165.4	166.0	164.6	393.3

Size	`select9`	`select9` pc	Hinted bsearch	`simple`	BitRankF	`darray`	Kim	Clark
1 Ki	45.0	60.0	34.5	41.3	82.3	44.0	47.5	166.4
16 Ki	45.3	63.2	37.8	35.8	103.4	44.0	48.2	179.8
256 Ki	45.3	63.8	38.2	36.1	121.4	44.0	50.0	195.7
4 Mi	58.8	76.6	49.6	42.6	144.4	56.0	98.4	223.0
64 Mi	245.7	263.7	213.6	145.0	344.6	185.0	320.2	485.1
1 Gi	367.5	383.1	316.5	230.2	978.2	323.0	557.5	599.1

Table 2: Nanoseconds per rank and select operations in densely populated (50%) bit arrays of increasing size. The space usage of rank structures is shown on their label; the space shown for Jacobson's structure is for the 1 Gi array (for smaller sizes, it grows significantly, as it happens for Clark's structure in Table 1: at size 2^{64}, it is still 37.5%; it becomes space-competitive with `rank9` beyond 2^{100} bits). As noted in [GGMN05], once out of the cache access time increase linearly due to the memory-address resolution process.

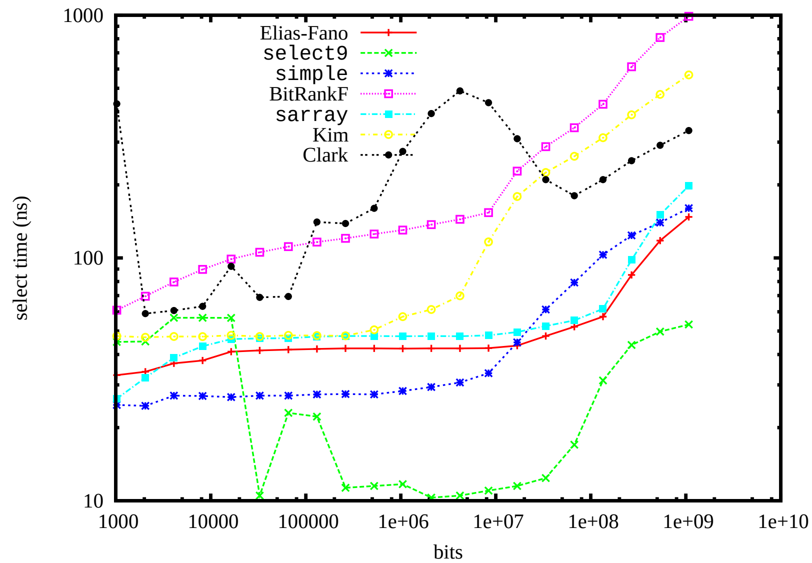


Size	`select9`	Hinted bsearch	`simple`	BitRankF	`darray`	Kim	Clark
1 Ki	45.2	36.2	38.2	71.0	44.0	47.4	247.6
16 Ki	51.3	45.5	120.5	102.0	148.0	47.9	122.4
256 Ki	33.8	51.6	20.7	121.0	52.0	48.0	158.9
4 Mi	35.0	62.6	28.1	143.9	34.0	73.9	341.1
64 Mi	161.3	224.3	101.7	343.9	119.0	295.1	301.8
1 Gi	209.7	366.5	144.8	988.4	195.0	510.9	434.9

Table 3: Nanoseconds per select operation in a densely (50%) populated bit array of increasing size with uneven bit distribution: almost all bits in the first half are zeroes, and almost all bits in the second half are ones. The "switch" effect typical of structures that change their strategy depending on the density is very visible. Note the poor performance on large arrays of methods based on binary search.

Size	`select9`	Hinted bsearch	`simple`	`darray`	Kim	Clark
1 Ki	56.25%	37.50%	25.00%	67.19%	94.53%	544073.24%
16 Ki	56.25%	37.50%	14.45%	28.81%	83.06%	34074.85%
256 Ki	56.20%	37.38%	63.96%	40.25%	80.93%	2184.99%
4 Mi	56.19%	37.37%	45.17%	43.23%	80.80%	192.81%
64 Mi	56.19%	37.38%	45.95%	43.61%	80.76%	68.30%
1 Gi	56.19%	37.38%	45.94%	43.60%	80.77%	60.52%

Table 4: Percentage of space occupied by various select structures in a densely (50%) populated uneven bit array (see Table 3).



Size	Elias–Fano	`select9`	`simple`	BitRankF	`sarray`	Kim	Clark
1 Ki	33.6	45.1	24.5	61.3	25.9	47.8	432.1
16 Ki	44.2	56.8	26.7	98.8	45.9	48.0	92.7
256 Ki	45.6	11.8	27.4	120.3	47.8	48.0	138.8
4 Mi	45.6	10.3	30.7	143.7	47.9	74.3	487.4
64 Mi	52.5	17.0	79.4	346.1	55.2	258.4	180.2
1 Gi	157.7	52.9	160.2	969.7	199.2	554.9	322.6

Table 5: Nanoseconds per select operation in bit arrays of increasing size with sparse (1%) bit population.

Select	Elias–Fano	`select9`	`simple`	`sarray`	Kim	Clark
1 Ki	84.77%	56.25%	25.00%	98.44%	45.31%	544073.24%
16 Ki	13.94%	50.39%	10.55%	15.33%	26.12%	34074.85%
256 Ki	9.45%	50.15%	9.01%	9.81%	22.65%	2184.99%
4 Mi	9.37%	50.13%	9.01%	9.64%	22.55%	192.81%
64 Mi	9.38%	50.13%	9.01%	9.64%	22.52%	68.30%
1 Gi	9.37%	50.13%	9.01%	9.63%	22.50%	60.52%

Table 6: Percentage of space occupied by various select structures in bit arrays of increasing size with sparse (1%) bit population. Note that the percentage shown for `select9` includes 25% for `rank9`. Elias–Fano and `sarray` do not require the original bit array (which contributes an additional 100% to the other structures).

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

[GGMN05] R. Gonzàlez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[Gol06] Alexander Golynski. Optimal lower bounds for rank and select indexes. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*, volume 4051 of *Lecture Notes in Computer Science*, pages 370–381. Springer, 2006.

[GRRR06] R.F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science (FOCS '89)*, pages 549–554, Research Triangle Park, North Carolina, 1989. IEEE Computer Society Press.

[KNKP05] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In Sotiris E. Nikoletseas, editor, *Proc. of the Experimental and Efficient Algorithms, 4th InternationalWorkshop*, volume 3503 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2005.

[Knu07] Donald E. Knuth. The Art of Computer Programming. Pre-Fascicle 1A. Draft of Section 7.1.3: Bitwise Tricks and Techniques, 2007.

[Lam75] Leslie Lamport. Multiple byte processing with full-word instructions. *Communications of the ACM*, 18(8):471–475, 1975.

[OS07] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007*. SIAM, 2007.