A Fast x86 Implementation of Select

Prashant Pandey, Michael A. Bender, and Rob Johnson

Stony Brook University, Stony Brook, NY USA
{ppandey,bender,rob}@cs.stonybrook.edu

Abstract

Rank and select are fundamental operations in succinct data structures, that is, data structures whose space consumption approaches the information-theoretic optimal. The performance of these primitives is central to the overall performance of succinct data structures.

Traditionally, the select operation is the harder to implement efficiently, and most prior implementations of select on machine words use 50-80 machine instructions. (In contrast, rank on machine words can be implemented in only a handful of instructions on machines that support POPCOUNT.) However, recently Pandey et al. [15] gave a new implementation of machine-word select that uses only four x86 machine instructions; two of which were introduced in Intel's Haswell CPUs.

In this paper, we investigate the impact of this new implementation of machine-word select on the performance of general bit-vector-select. We first compare Pandey et al.'s machine-word select to the state-of-the-art implementations of Zhou et al. [20] (which is not specific to Haswell) and Gog et al. [8] (which uses some Haswell-specific instructions). We exhibit a speedup of $2\times$ to $4\times$.

We then study the impact of plugging Pandey et al.'s machine-word select into two state-of-the-art bit-vector-select implementations. Both Zhou et al.'s and Gog et al.'s select implementations perform a single machine-word select operation for each bit-vector select. We replaced the machine-word select with the new implementation and compared performance. Even though there is only a single machine-word select operation, we still obtained speedups of 20% to 68%. We found that the new select not only reduced the number of instructions required for each bit-vector select, but also improved CPU instruction cache performance and memory-access parallelism.

1998 ACM Subject Classification "E.1 DATA STRUCTURES", "E.2 DATA STORAGE REPRESENTATIONS"

Keywords and phrases Succinct data structures, Rank and Select operation

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

A ***succinct data structure*** consumes an amount of space that is close to the information-theoretically optimal. More precisely, if Z denotes the information-theoretically optimal space usage for a given data-structure specification, then a succinct data structure would use $Z + o(Z)$ space. There is much research on how to replace traditional data structures with succinct alternatives [2–4, 7, 13, 16, 17], especially for memory-intensive applications, such as genetic databases, newspaper archives, dictionaries, etc. Researchers have proposed numerous succinct data structures, including compressed suffix arrays [10], FM indexes [5], and wavelet trees [13].

Two basic operations—namely ***rank*** and ***select*** [12]—are commonly used for navigating within succinct data structures. For bit vector $B[0, \ldots, n-1]$, $\textsc{rank}(j)$ returns the number of 1s in prefix $B[0, \ldots, j]$ of B; $\textsc{select}(r)$ returns the position of the rth 1, that is, the smallest index j such that $\textsc{rank}(j) = r$. For example, for the 12-bit vector $B[0, \ldots, 11] =$ `100101001010`, $\textsc{rank}(5) = 3$, because there are three bits set to one in the 6-bit prefix $B[0, \ldots, 5]$ of B, and $\textsc{select}(4) = 8$, because $B[8]$ is the fourth 1 in the bit vector.

Researchers have proposed many ways to improve the empirical performance of rank and select [8, 9, 14, 18, 20] because faster implementations of rank and select yield faster succinct data structures. Many succinct data structures are asymptotically optimal, but have poor performance in practice due to the constants involved in bit-vector rank and select operations [18].

This paper focuses on a new implementation of select on machine words and its performance impact on general bit-vector select operations.

Implementing Select

Most practical select implementations chunk the bit vector into ***basic blocks***. They also maintain one or more levels of indexes, called ***superblocks***, which identify the basic block containing the jth bit, for some predetermined values of j.

The select operation proceeds in two phases. In the first phase SELECT(j) does a lookup in the superblocks and finds the basic block that contains the jth one. In the second phase the algorithm performs a select operation on the basic block. If the basic block is larger than a machine word, then the algorithm uses POPCOUNT sequentially to find the machine word that contains the jth bit of the vector. It then performs a select on that machine word to find the exact location of the jth bit in the vector.

Implementing select on a machine word turns out to be surprisingly complex, when only standard operations such as shift, mask, multiply, addition, etc. are available. Therefore, the select operation on machine words represents a significant component of the time required to perform select, even on large bit vectors.

For machine words, the select operation has been harder to implement than the rank operation. The rank operation on a machine word can be implemented using a dedicated machine instruction, POPCOUNT, which counts the number of 1s in a machine word. In contrast, until recently, the best known implementations of select on machine words used 50–80 machine instructions [20]. Most prior implementations use broadword programming [14, 18], SSE4 instructions [8] or a combination of broadword programming and the POPCOUNT instruction [20].

Recently, Pandey et al. [15] gave a new implementation of machine-word select, which we call PTSelect. PTSelect uses only four x86 machine instructions, two of which were introduced in Intel's Haswell CPUs in 2013. One instruction, PDEP, deposits bits from one operand in locations specified by the bits of the other operand. The second instruction, TZCNT, returns the number of trailing zeros in its argument. Although these two instructions did not exist a few years ago, they are widely available today.

Results

- We perform an experimental study of how PTSelect, the new implementation of the select operation on 64-bit words proposed by Pandey et al. [15], affects select on large bit vectors.
- We run two different benchmarks. First we compare PTSelect to CS-Poppy [20] and SDSL [8] on machine-word select.
- Second, we replace the machine-word select implementations in CS-Poppy and SDSL with PTSelect. We then evaluate the performance gain for select on large bit vectors.
- We show the performance gain as a function of the bit-vector size.
- Our experiments show that PTSelect runs $2\times$—$4\times$ faster than the state-of-the-art implementation [20] on 64-bit machine words.
- When we replace the machine-word select implementation with PTSelect on large bit vectors we see a performance gain of 20% to 68%.
- We found that the new select not only reduced the number of instructions required for each bit-vector select, but also improved CPU instruction cache performance and memory-access parallelism.

Algorithm 1 Algorithm for determining the position of the jth 1 in a machine word.

```
1: function PTSELECT($x$, $j$)
2:     $i \leftarrow \textsc{ShiftLeft}(1, j)$
3:     $p \leftarrow \textsc{pdep}(i, x)$
4:     return $\textsc{tzcnt}(p)$
```

2 How Select is Being Implemented on Large Bit Vectors

This section reviews the general bit-vector select implementations in CS-Poppy [20] and SDSL [8] and explains the role that machine-word select performs in these algorithms. We then discuss the differences in the implementations of CS-Poppy and SDSL.

Both CS-Poppy and SDSL use a position-based sampling approach to implement bit-vector select. In position-based sampling, the bit-vector maintains a side table $S[i]$, where $S[i] = \textsc{select}(ki)$ for some constant k. The algorithm thus knows that, for any y, $S[\lfloor y/k \rfloor] \leq \textsc{select}(y) < S[\lfloor y/k \rfloor + 1]$. It may then use additional auxiliary data structures to further narrow down the location of the yth one to a single machine word. After finding the target machine word, it performs a machine-word select to find the exact location of the yth one in the bit vector.

2.1 Select in CS-Poppy

CS-Poppy groups bits into 512-bit *basic blocks*, groups each group of 4 basic blocks into a *lower block*, and groups 2^{21} lower blocks into an *upper block*. It maintains a table L_0 of the rank of the first bit of each upper block. A select operation begins by searching in L_0 for the upper block containing the target bit. The paper says this is done using binary search; the implementation performs a linear scan. For each upper block, CS-Poppy maintains a table S as described above, which it uses to find a lower block B near the target bit. CS-Poppy then consults a table L_1 of the rank of the first bit of each lower block. It uses L_1 to scan forward from B until it finds the lower block B' that actually contains the target bit. It then consults a table L_3 of the popcount of each basic block to scan through the basic blocks of B' until it finds the basic block that contains the target bit. It then performs a linear scan of the words in the basic block, using POPCOUNT to find the machine word that contains the target bit. Finally, it performs a machine-word select to find the precise location of the target bit.

CS-Poppy optimizes cache performance by using 512-bit basic blocks (which occupy a single cache line), and by storing L_1 and L_2 in a packed, interleaved format so that accessing L_2 does not incur any additional cache miss after accessing the corresponding entry in L_1. See the paper for details [20].

2.2 Select in SDSL

SDSL divides the bit vector in superblocks by storing position of every 4096-th one in a table S as described above. If the size of a superblock (i.e., difference in the index of $k4096$-th one and $(k+1)4096$-th one, for some k) is larger than or equal to $log^4 n$, where n is the number of bits in the bit vector, then it is called *long* and SDSL maintains another level of positions by storing the positions of all 4096 ones. If the size of a superblock is smaller than $log^4 n$ then it is called *short*, and SDSL maintains another level of positions by storing the positions of every 64-th one.

To perform $\textsc{select}(y)$, the algorithm first performs a lookup in superblocks. If the yth bit occurs in a long superblock then it can answer by a look up in the next level. Otherwise, if the yth bit occurs in a short superblock, it looks up in the next level to determine the nearest machine word. The algorithm then sequentially performs POPCOUNT to determine the target 64-bit word in which the yth bit occurs. It then performs a select on the target 64-bit word.

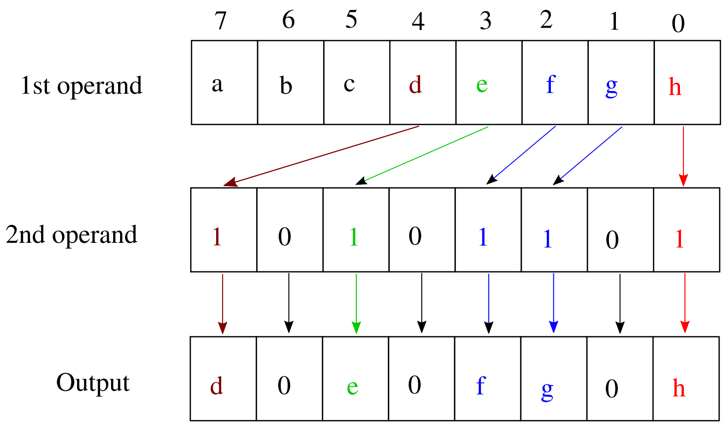


Figure 1 A simple example of pdep instruction [11]. The 2nd operand acts as a mask to deposit bits from the 1st operand.

3 PTSelect

This section reviews Pandey, et al.'s PTSelect implementation, shown in Algorithm 1. To select the jth bit from a 64-bit word x, the algorithm first loads the constant 1 into a register, shifts it left by j, then invokes PDEP and TZCNT.

We now explain the PDEP and TZCNT instructions, both of which were introduced in Intel's Haswell line of CPUs.

PDEP deposits bits from one operand in locations specified by the bits of the other operand, as illustrated in Figure 1. If $p = \text{PDEP}(v, x)$, then the ith bit of p is given by

$$p_i = \begin{cases} v_j & \text{if } x_i \text{ is the } j\text{th 1 in } x, \\ 0 & \text{otherwise.} \end{cases}$$

TZCNT returns the number of trailing zeros in its argument. If B is a 12-bit vector such that $B[0, 11] =$ `110010100000` then $\text{TZCNT}(B) = 5$.

The PTSelect algorithm works because performing $\text{PDEP}(2^j, x)$ produces a 64-bit integer p with a single bit set—in the same position as x's jth bit. TZCNT then finds the position of this bit by counting the number of trailing zeros in p.

4 Evaluation

In this section we evaluate PTSelect, the new machine-word select implementation proposed by Pandey et al. [15]. We compare PTSelect against machine-word select implementations in Zhou at al's CS-Poppy [19] (which is an optimized implementation of combined sampling [14]) and Gog et al.'s SDSL [6] (which is an implementation of Clark et al. [1] and broadword programming). We then measure the impact of PTSelect on the performance of the overall bit-vector select algorithms in these implementations.

We address the following questions about the performance of PTSelect:

- How does PTSelect compare to machine-word select implementations in CS-Poppy and SDSL on 64-bit words in cache?
- How does PTSelect compare to machine-word select implementations in CS-Poppy and SDSL on 64-bit words not in cache?
- How does PTSelect affects the performance of bit-vector select in CS-Poppy and SDSL.

Implementation	Time (ns)
CS-Poppy	8.11
SDSL	4.23
PTSelect	2.07

Table 1 Time to perform a machine-word select on an in-cache 64-bit machine word by CS-Poppy, SDSL, and PTSelect.

4.1 Experimental Setup

To answer the above questions, we perform three different benchmarks. First we evaluate the performance of PTSelect and machine-word select implementations in CS-Poppy and SDSL on a single 64-bit machine word. Second we evaluate the performance of PTSelect and machine-word select implementations in CS-Poppy and SDSL on 64-bit machine words drawn randomly from a bit vector. Third we replace the machine-word select implementations in CS-Poppy and SDSL with PTSelect and evaluate the performance for select operations on large bit vectors.

For our experiments we used the same benchmark suite that is used in Zhou et al.'s paper [19]. To evaluate SDSL on the same benchmark, we took the bit vector select implementation from the SDSL library [6] and added it to the benchmark suite. Our version of the benchmarking suite is available at `https://github.com/splatlab/rankselect`.

We performed microbenchmarks to measure how fast PTSelect runs on machine words compared to the machine-word select implementations in CS-Poppy and SDSL. In the first benchmark we perform 10M select operations on a single machine word with random rank values and report the total time. Because we perform select on a single machine word there are essentially no cache misses during the experiment.

For the second benchmark, we measure the time required to perform 10M machine-word select on a word chosen randomly from a bit vector. Thus, for large bit vectors, the select is likely to incur a cache miss loading the input word. This benchmark measures the performance difference that different machine-word select implementations yield in a realistic workload of essentially random selects out of a bit vector.

For the third benchmark, we perform 10M bit-vector selects. This benchmark measures the total time to perform the select, including traversing data structures, performing a machine-word select, and cache misses.

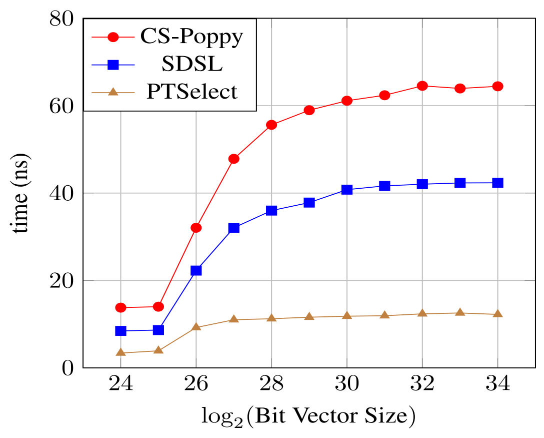
We perform these experiments with bit vector sizes from 2^{24} to 2^{34}. For each of the benchmarks, we perform experiments with three different densities of the bit vector, 90%, 50%, and 10%. We perform 10 iterations of each experiment and report the average time (in nanoseconds) to perform a single operation.

We have also collected some hardware stats in order to analyze our evaluation results. We have used two tools, *perf* and *intel VTune* to collect stats. We used *perf* to collect the number of instruction-cache misses and the number of instructions. We used *intel VTune* to collect the average number of in-flight memory read requests.

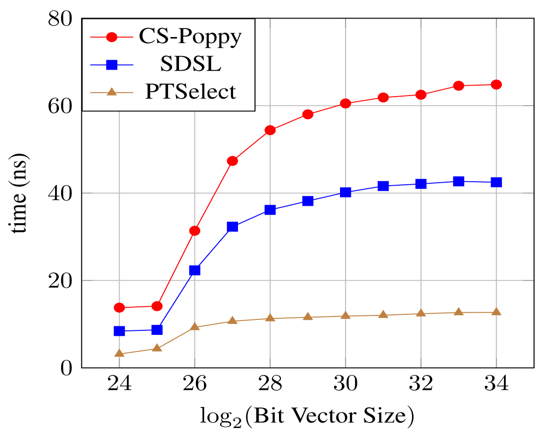
All experiments were performed on an Intel(R) Core(TM) i7-6700HQ CPU (@ 2.60GHz with 4 cores and 6MB LLC) running Ubuntu x86_64 4.4.0-59-generic.

4.2 In-register machine-word select

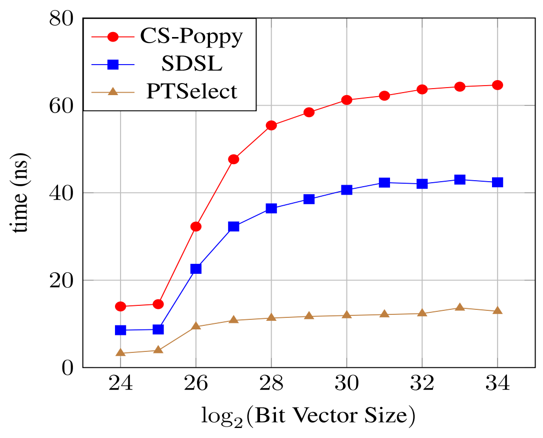
Table 1 shows the time taken to perform select on in-cache 64-bit words. PTSelect is $4\times$ faster than the CS-Poppy machine-word select and $2\times$ faster than the SDSL machine-word select.



(a) Machine word density 90%



(b) Machine word density 50%



(c) Machine word density 10%

Figure 2 Performance of select operation on machine words with density 90%, 50%, and 10%. (Lower is better.)

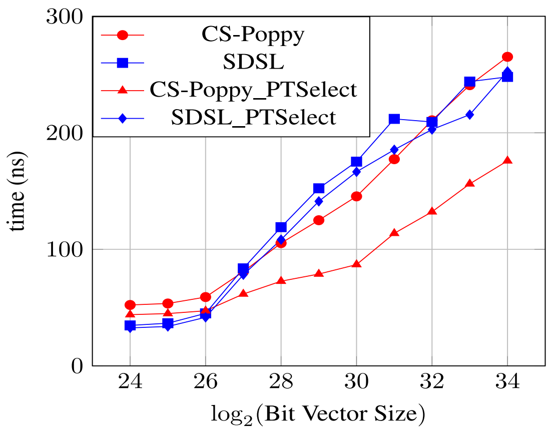
4.3 Machine-word select

Figure 2 shows the average time taken per select operation by CS-Poppy, SDSL, and PTSelect for select operations on machine words chosen randomly from a bit vector.

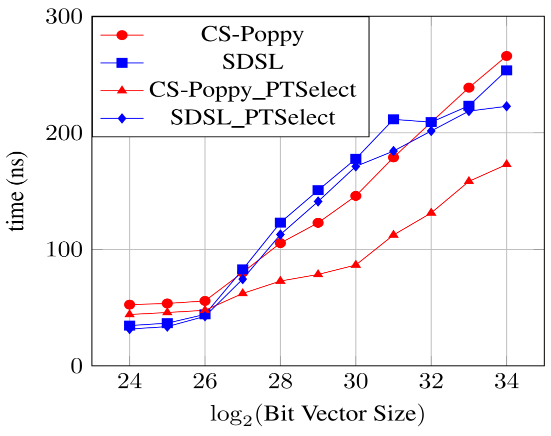
PTSelect is 2.2×–6× faster than CS-Poppy and SDSL. CS-Poppy machine-word select is the slowest for any bit vector size. For larger bit vectors, PTSelect is 5× faster than CS-Poppy and 3× faster than SDSL.

We explain these results as follows. When the bit vector is small, cache misses are relatively rare, so the cost of each select is dominated by the computational costs, which are shown in Table 1. As the bit vector grows, the time to perform a machine-word select becomes dominated by the time required to load the queried machine word into cache.

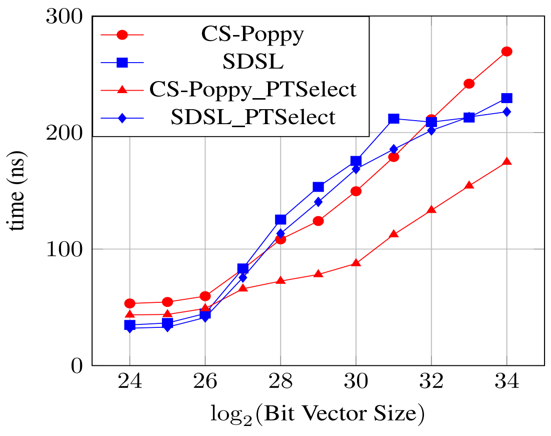
However, this presents one puzzle: if the time to perform a machine-word select within a large bit vector is dominated by the time to resolve a cache miss, why is PTSelect faster than SDSL, which is in turn faster than CS-Poppy? Shouldn't they all take roughly the same amount of time, i.e. the time required to serve the cache miss?



(a) Bit vector density 90%



(b) Bit vector density 50%



(c) Bit vector density 10%

Figure 3 Performance of select operation on large bit vector with densities 90%, 50%, and 10%. (Lower is better.)

The reason they do not all take the same amount of time is *memory parallelism*. Modern CPUs can issue multiple memory reads in parallel, increasing throughput. Figure 4 shows the average level of memory parallelism observed for each algorithm during our machine-word select benchmark. PTSelect has almost 3× the parallelism of CS-Poppy, and about 2× the parallelism of SDSL. This largely explains the performance difference between the different machine-word select algorithms seen for large bit vectors in Figure 2.

And why do the different machine-word select implementations enable different levels of memory parallelism? Modern CPUs maintain a relatively large window of outstanding instructions, any of which can be issued as soon as its inputs are available. Each iteration of our benchmark loop is independent of previous iterations, so the CPU can potentially execute two or more iterations in parallel, if they fit within its window of outstanding instructions. Thus, a shorter implementation of select will enable more iterations to fit in the CPU's instruction window, enabling greater parallelism. Each iteration will induce roughly one cache miss so, as a result, a shorter select implementation will enable greater memory parallelism.

Implementation	Instruction count
CS-Poppy	80
SDSL	49
PTSelect	12

Table 2 The number of instructions in each implementation of machine-word select. These counts include function prologues and epilogues and code for loading the arguments from bit-vector, in addition to the core instructions for performing the machine-word select.

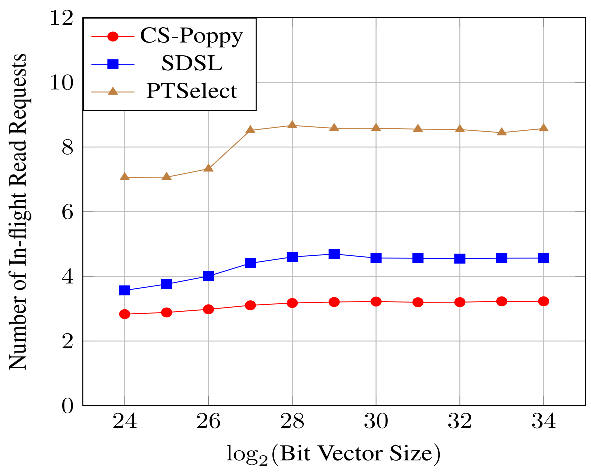


Figure 4 The average number of in-flight read requests per cycle when performing machine-word select operations on increasing bit vector sizes for CS-Poppy, SDSL, and PTSelect.

This is exactly what we see in the benchmark. Table 2 shows the number of instructions in each select implementation. All the implementations consist of straight-line code with no branches. CS-Poppy consists of 80 instructions and averages 3 concurrent memory accesses, suggesting an instruction window of about 240 instructions. SDSL consists of 49 instructions and averages 4 concurrent memory accesses, suggesting an instruction window of about 200 instructions. PTSelect contains only 12 instructions and averages 8 concurrent memory accesses, which would suggest a window size of only about 100 instructions. We suspect that PTSelect is actually being bottlenecked by the number of outstanding memory requests supported by the CPU.

4.4 Bit-vector select

Figure 3 shows the average time taken per bit-vector select by CS-Poppy, SDSL, CS-Poppy with PTSelect and SDSL with PTSelect.

Both CS-Poppy and SDSL are faster when we replace their default machine-word select implementation with PTSelect. CS-Poppy with PTSelect performs 20%–68% faster than traditional CS-Poppy. SDSL with PTSelect performs 2%–15% faster than traditional CS-Poppy.

5 Conclusion

This paper shows that having an efficient implementation of machine-word select leads to a faster bit-vector select. In our evaluation we found that PTSelect is the fastest machine-word select implementation. CS-Poppy with PTSelect is the fastest bit-vector select for large bit vectors and SDSL with PTSelect is the fastest bit-vector select for small bit vectors.

Even though the machine-word select is invoked only once during a bit-vector select (irrespective of the size of the bit vector) the performance gain by using PTSelect increases with increasing bit-vector sizes. This is because the small size of PTSelect enables the CPU to execute more selects in parallel, increasing memory-level parallelism, improving performance for bit vectors that do not fit in cache.

We believe that there may be additional opportunities to optimize succinct data structure implementations using x86 bit-manipulation instructions. Furthermore, by making succinct data structures that match the performance of conventional data structures, we can bring succinct data structures from theory into practice.

Acknowledgments. We gratefully acknowledge David Anderson for providing the benchmarking suite for our comparisons from Zhou et al.'s CS-Poppy. We thank Mike Ferdman for suggesting that memory-level parallelism might explain the performance we observed in our machine-word select benchmark.

We gratefully acknowledge support from NSF grants IIS-1247726, IIS-1251137, CNS-1408695, CCF-1439084, and CCF-1617618, and from Sandia National Laboratories.

References

1 David Clark. *Compact Pat nees*. PhD thesis, PhD thesis, University of Waterloo, 1998.

2 J Shane Culpepper, Gonzalo Navarro, Simon J Puglisi, and Andrew Turpin. Top-k ranked document search in general text databases. In *European Symposium on Algorithms*, pages 194–205. Springer, 2010.

3 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

4 Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

5 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

6 Simon Gog. Succinct data structure library. `https://github.com/simongog/sdsl-lite`, 2017. [online; accessed 01-Feb-2017].

7 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

8 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

9 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

10 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

11 Yedidya Hilewitz and Ruby B Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *Application-specific Systems, Architectures and Processors, 2006. ASAP'06. International Conference on*, pages 65–72. IEEE, 2006.

12 Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

13 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys (CSUR)*, 46(4):52, 2014.

14 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pages 295–306. Springer, 2012.

15 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proc. 2017 International Conference on Management of Data (SIGMOD)*, May 2017. to appear (draft available at `http://www3.cs.stonybrook.edu/~rp/tech_reports/sbcstr-c6ff764fdd8f9d2b5ea3b31972a787bc/report.pdf`).

16 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

17 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

18 Sebastiano Vigna. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer, 2008.

19 Dong Zhou. Rankselect benchmark. `https://github.com/efficient/rankselect`, 2017. [online; accessed 01-Feb-2017].

20 Dong Zhou, David G Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *International Symposium on Experimental Algorithms*, pages 151–163. Springer, 2013.