The final version appears in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, Rome, Italy, June 2013

Space-Efficient, High-Performance Rank & Select Structures on Uncompressed Bit Sequences

Dong Zhou, David G. Andersen, Michael Kaminsky[†]
Carnegie Mellon University, [†]Intel Labs

Abstract. Rank & select data structures are one of the fundamental building blocks for many modern *succinct data structures*. With the continued growth of massive-scale information services, the space efficiency of succinct data structures is becoming increasingly attractive in practice. In this paper, we re-examine the design of rank & select data structures from the bottom up, applying an architectural perspective to optimize their operation. We present our results in the form of a recipe for constructing space and time efficient rank & select data structures for a given hardware architecture. By adopting a *cache-centric* design approach, our rank & select structures impose space overhead as low as the most space-efficient, but slower, prior designs—only 3.2% and 0.39% extra space respectively—while offering performance competitive with the highest-performance prior designs.

1 Introduction

Rank & select data structures [6] are one of the fundamental building blocks for many modern *succinct data structures*. Asympototically, these data structures use only the minimum amount of space indicated by information theory. With the continued growth of massive-scale information services, taking advantage of the space efficiency of succinct data structures is becoming increasingly attractive in practice. Examples of succinct structures that commonly use rank & select include storing monotone sequences of integers [2,3] and binary or n-ary trees [6,1]. These structures in turn form the basis for applications such as compressed text or genome searching, and more.

For a zero-based bit array B of length n, the two operations under consideration are:

1. `Rank`(x) - Count the number of 1s up to position x;
2. `Select`(y) - Find the position of the y-th 1.

More formally, let B_i be the i-th bit of B, then

$$\texttt{Rank}(x) = \sum_{0 \le i < x} B_i, \quad 1 \le x \le n$$

and

$$\texttt{Select}(y) = \min\{\ k \mid \texttt{Rank}(k) = y\ \}, \quad 1 \le y \le \texttt{Rank}(n)$$

For example, in the bit array `0,1,0,1,0`, using zero-based indexing, `Rank(2)=1`, and `Select(1)=1`.

In this paper, we consider the design of rank & select data structures for large in-memory bit arrays—those occupying more space than can fit in the CPU caches in modern processors, where n ranges from a few million up to a few tens of billions. We present our results in the form of a recipe for constructing space and time efficient rank & select data structures for a given hardware architecture. Our design, like several practical implementations that precede it [10,5,9], is not strictly optimal in an asymptotic sense, but uses little space in practice on 64-bit architectures.

The core techniques behind our improved rank & select structures arise from an aggressive focus on cache-centric design: It begins with an extremely small (and thus, cache-resident) first-layer index with 64-bit entries. This index permits the second-layer index to use only 32-bit entries, but maintains high performance by not incurring additional cache misses. This first-layer index is followed by an interleaved second and third layer index that is carefully sized so that accessing both of these indices requires only one memory fetch. The result of this design is a structure that simultaneously matches the performance of the fastest available rank & select structure, while using as little space as the (different) most space-efficient approach, adding only 3.2% and 0.39% space overhead for rank and select, respectively.

2 Design Overview & Related Work

Before we dive into the detailed design of our rank & select data structures, we first provide an overview of previous approaches, identify common design frameworks shared among them, and examine their merits and drawbacks. Because rank & select are often implemented in different ways, we discuss them separately.

2.1 Rank

For rank, almost all previous approaches embrace the following design framework:

1. Determine the size of the *basic block*, along with an efficient way to count the number of bits inside a basic block. Because the basic block is the lowest level in the rank structure, we should be able to do counting directly upon the original bit array.
2. Design an *index*, with one or multiple layers, that provides the number of 1s preceding the basic block in which x is located. Each index entry maintains aggregation information for a group of consecutive basic blocks, or *superblocks*.

Figure 1 illustrates a typical two-layer rank structure. In this example, basic blocks have a size of 8 bits. Entries in the first layer index are absolute counts, while entries in the second layer index count relative to the superblock start, rather than the very beginning of the bit array. Whenever a query for `rank`(x) comes in,

1. First, look in the first layer index to find p, the number of 1s preceding the superblock into which x falls.
2. Second, look in the second layer index to find q, the number of 1s within that superblock that are to the left of the basic block into which x falls.
3. Finally, count the number of 1s to the left of x within that basic block, r.

The answer to `rank`(x) is then $p + q + r$.

To demonstrate the generality of this design framework, we summarize several representative approaches, along with our rank structure, in Table 1. `RG 37` is a variant

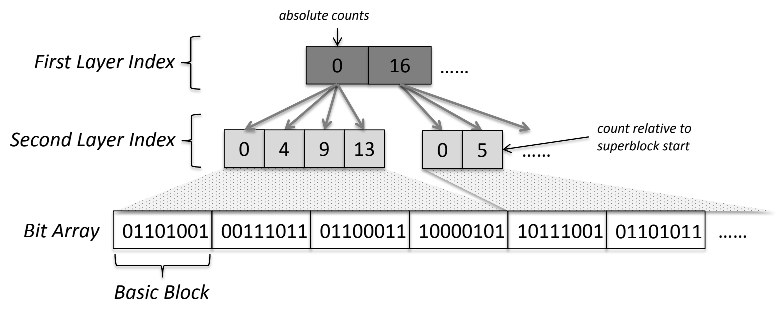


Fig. 1: Generalized Rank Structure.

Approach	Basic Block Size	In-block Counting Method	Index Design	Space Overhead	Max Supported Size
Classical solution [5]	$\lfloor \log(n)/2 \rfloor$ bits	Precomputed table	Two-layer index	66.85%	2^{32}
`RG 37` [5]	256 bits	Precomputed table	Two-layer index	37.5%	2^{32}
`rank9` [10]	64 bits	Broadword programming	Two-layer index	25%	2^{64}
`combined sampling` [9]	1024 bits	Precomputed Table	One-layer index	3.125%	2^{32}
Ours (`poppy`)	512 bits	`popcnt` instruction	Three-layer index	3.125%	2^{64}

Table 1: Previous & Our Rank Structures.

of the classical constant-time solution proposed by González et al. [5], and adds 37.5% extra space above the raw bit array. `rank9` [10] employs broadword programming [7] to efficiently count the number of one bits inside a 64-bit word[1], and stores the first and second layer of index in an *interleaved form*—each first layer index entry is followed by its second layer entries, which reduces cache misses. `combined sampling` [9] explores the fact that the space overhead is inversely proportional to the size of the basic block, and achieves low space overhead ($\sim 3\%$) by using 1024-bit basic blocks. However, this space efficiency comes at the expense of performance. It is roughly 50%–80% slower than `rank9`. Therefore, our goal is to match the performance of `rank9` and the space overhead of `combined sampling`.

Notice that except for `rank9`, all of the previous rank structures can only support bit arrays that have up to 2^{32} bits. However, as the author of `rank9` observes, efficient rank & select structures are particularly useful for extremely large datasets: a bit array of size 2^{32} substantially limits the utility of building compressed data structures based on rank & select. Unfortunately, naively extending existing structures to support larger bit arrays by replacing 32 bit counters with 64 bit counters causes their space overhead to nearly double.

From the above overview, we identify three important features for a rank structure:

1. **Support bit arrays with up to 2^{64} bits.**
2. **Add no more than 3.125% extra space.**
3. **Offer performance competitive to the state-of-art.**

Section 3.1 explores the design of our new rank structure, called `poppy`, which fulfills all three requirements.

[1] Broadword programming, also termed as "SWAR" (SIMD Within A Register), can count the number of one bits in $O(\log d)$ instructions, where d is the number of bits in the word. The latest version of `rank9` replaces broadword programming with `popcnt` instruction.

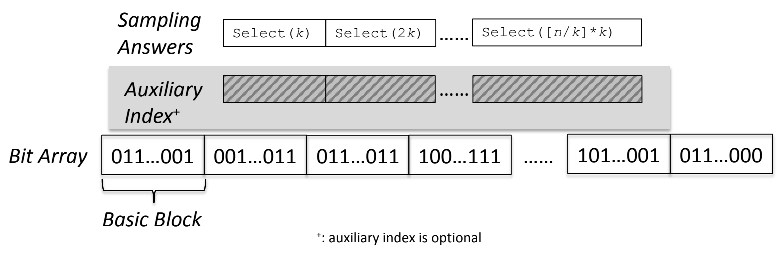


Fig. 2: Generalized Position-Based Select Structure.

2.2 Select

Two general approaches are used to implement select. One is *rank-based selection*, and the other is *position-based selection*. For both methods, the first step is the same:

1. Determine the size of the *basic block*, and an efficient way to select within a basic block. This size need not be the same as the basic block size for rank, but making them the same is a common design choice.

For *rank-based selection*, the second step is:

2a. Design an *index* with one or multiple layers that identifies the location of the basic block in which the x-th one bit is located. This index is similar to the index for rank, but the demand imposed on it is different. In rank, we know exactly which entry is needed. For example, given a rank structure with 8-bit basic blocks, bit 15 is always in the second basic block. However, the 15-th *one bit* might be located in the 10000-th basic block! Therefore, in rank-based select, we must find the correct entry by *searching* (most commonly, by binary searching). These two distinct access patterns give us an intuitive understanding of why select is more difficult than rank. Although the index for select is similar to that for rank, they are not necessarily the same.

And for *position-based selection*, the second step is:

2b. Store a sampling of `select` answers and possibly an auxiliary index. Using these two structures, we can reach a position that is very close to the basic block in which the x-th one bit is located. Then, scan sequentially to find the correct basic block.

Figure 2 presents a typical position-based select structure, which stores select results for every k ones. To answer `select`(y), we first find the largest j such that $jk \leq y$. Because `select`(jk) is stored in a precomputed table, we can obtain the position of the jk-th one bit by a lookup in that table. Then, we locate the basic block containing the y-th one bit with or without the help of an auxiliary index. After finding the target basic block, we perform an in-block select to find the correct bit position.

Table 2 lists several previous approaches for select. The space overhead, *excludes* the space occupied by the rank structure, though several select structures rely on their corresponding rank structure to answer queries. Similar to rank, three features are desirable for a select index:

Approach	Type	Basic Block Size	In-block Selection Method	Space Overhead	Max Supported Size
Clark's structure [1]	Position-based	$\lceil \log n \rceil$ bits	Precomputed Table	60%	2^{32}
Hinted bsearch [10]	Rank-based	64 bits	Broadword programming	$\sim$ 37.38%	2^{64}
`select9` [10]	Position-based	64 bits	Broadword programming	$\sim$ 50%	2^{64}
`simple select` [10]	Position-based	64 bits	Broadword programming[2]	9.01%-45.94%	2^{64}
`combined sampling` [9]	Position-based	1024 bits	Byte-wise table lookup + bit-wise scan	$\sim$ 0.39%	2^{32}
Ours (`cs-poppy`)	Position-based	512 bits	`popcnt` + broadword programming	$\sim$ 0.39%	2^{64}

Table 2: Previous Select Structures.

1. **Support bit arrays with up to 2^{64} bits.**
2. **Add no more than 0.39% extra space.**
3. **Offer performance competitive to the state-of-art.**

Section 3.2 explores the design of our new select structure, called `combined sampling with poppy` or `cs-poppy` for short, which fulfills all three requirements.

3 Design

In light of the above observations, we now present our design recipe for a rank & select data structure. Like most previous solutions, we use a hierarchical approach to rank & select. Our recipe stems from three underlying insights from computer architecture:

For large bit arrays, the overall performance is strongly determined by cache misses. In a bit array occupying hundreds of megabytes of space, it is necessary to fetch at least one block from memory into the cache. Thus, optimizing the computation to be much faster than this fetch time does not provide additional benefit. A fetch from memory requires approximately 100ns, enough time to allow the execution of hundreds of arithmetic operations.

Parallel operations are cheap. Executing a few operations in parallel often takes only modestly longer than executing only one. This observation applies to both arithmetic operations (fast CPUs execute up to 4 instructions at a time) and memory operations (modern CPUs can have 8 or more memory requests in flight at a time).

Optimize for cache misses, then branches, then arithmetic/logical operations. There is over an order of magnitude difference in the cost of these items: 100ns, 5ns, and $< \frac{1}{4}$ns, respectively. A related consequence of this rule is that it is worth engineering the rank/select structures to be cache-aligned (else a retrieval may fetch two cachelines), and also to be 64-bit aligned (else a retrieval may cost more operations).

In the rest of this section, we describe our design as optimized for recent 64-bit x86 CPUs. When useful, we use as a running example the machine from our evaluation (2.3 GHz Intel Core i7 "Sandy Bridge" processor, 8 MB shared L3 cache, 8 GB of DRAM).

3.1 Rank

Basic Block for Rank The basic block is the lowest level of aggregation. Within a basic block, both `Rank` and `Select` work by counting the bits set up to a particular position

[2] As `rank9`, the latest version of `simple select` uses `popcnt` + broadword programming.

Method	Time (ms)
Precomputed table (byte-wise)	729.0
`popcnt` instruction	191.7
SSE2	336.0
SSSE3	237.7
Broadword programming	798.9

Table 3: Performance for different methods of popcounting a 64M bit array 300 times.

(often referred as *population count* or *popcount*), without using any summary information. Both theoretical analysis as well as previous approaches demonstrate that the space overhead of rank & select is inversely proportional to the size of the basic block. Larger basic blocks of bits mean that fewer superblocks are needed in the index. Meanwhile, excessively enlarging the size of the basic block degrades performance, because operating on larger blocks requires more computation and more memory accesses, which are extremely expensive. Specifically, the number of memory accesses grows *linearly* as the size of the basic block increases. Therefore, algorithm implementers should focus most of their effort on finding techniques to *efficiently* increase the number of bits that can be processed at the lowest level with no auxilary information.

Previous work showed that we can set this size to 32 bits and perform popcount using lookups in a precomputed table [5,9], or set the size to 64 bits and use the broadword programming bit-hacking trick to implement popcount in $O(\log d)$ instructions where d is the number of bits in the word [10]. Other choices include using the vector SSE instructions (SSE2), the `PSHUFB` instruction (SSSE3) which looks up 4 bits at a time in a table in parallel, or as proposed recently by Ladra et al. [8], the `popcnt` instruction which is available in newer Intel processors (Nehalem and later architectures).

We ran microbenchmarks and measured the performance of each method. The microbenchmark creates a bit array of 64M bits and measures the performance of each method by popcounting the entire bit array 300 times. Because we count the number of one bits over the entire array, multiple *popcounts* can be in flight at the same time. The results (Table 3) show that the `popcnt` instruction is substantially faster than other approaches.

Next we must choose the basic block size. One straightforward design is to use 64-bit basic blocks, as in the design by Vigna [10]. However, as we noted above, larger basic blocks reduce the space overhead of the index; furthermore, executing several operations in parallel often takes only modestly longer than executing a single instruction. We therefore want to find the largest *effective* size. We call a size effective if moving up to that size yields performance benefit from parallel operations. If, instead, when we double the size of the basic block, the amount of time to popcount also doubles, this is a strong indicator that it is time to stop increasing the block size. Table 4 shows the performance of popcounting different basic block sizes using 10^8 random positions over a bit array with 2^{32} bits.

Before 512 bits, each doubling of the basic block slows execution by less than 2x, which implies that 512 is the right answer to the question. This also matches our expectation from a computer architecture perspective: the overwhelming factor in performance is cache misses. The size of a cache line is 512 bits. Hence, for well-aligned bit arrays, popcounting 512 bits leads to exactly one cache miss. In short, not only can

Size (bits)	Time (seconds)	# of cache misses
64	0.13	1
128	0.19	1
256	0.30	1
512	0.50	1
1024	0.99	2
2048	2.01	4

Table 4: Performance for popcounting 10^8 randomly chosen blocks of increasing sizes.

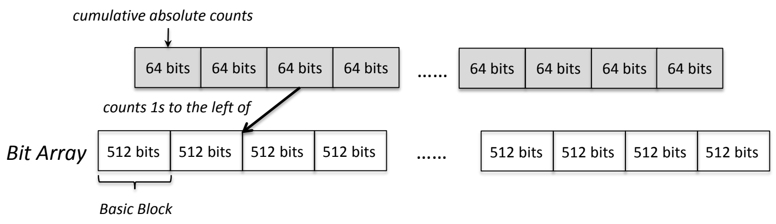


Fig. 3: Strawman Design of Rank Structure.

we popcount 512 bits extremely quickly, but doing so does not steal memory bandwidth from other operations. This choice contributes greatly to the speed of our space-efficient design for rank & select data structures.

Ladra et al. [8] also observed that varying the basic block size of the auxiliary data structure for rank and select offers a space/time tradeoff, which they can leverage to improve their space overhead. Here, we provide additional insight about how to best use their observation: by incorporating knowledge about the underlying memory hierarchy, our proposed guideline can help algorithm implementors understand how to make this space/time tradeoff.

Layered Index With popcount efficiently supporting blocks of 512 (2^9) bits, we have considerable flexbility in designing the index without sacrificing space. For example, an index that supports up to 4 billion bits (2^{32}) could simply directly index each 512-bit basic block, adding only 6.25% extra space. However, efficient rank & select data structures are particularly useful for extremely large datasets, and thus we would like to support a larger bit array.

Strawman Design. The strawman design (Figure 3) is to directly index each 512-bit basic block using a 64-bit counter to store the number of one bits to the left of that basic block. This solution offers good performance (roughly two cache misses per query: one for looking up in the rank structure, the other for examing bit vector itself), and adds 12.5% extra space.

To reduce the space overhead, we adopt two optimizations, each of which halves the index space, as illustrated in Figure 4.

Optimization I: 64 bit L0. In order to support more than 4 billion bits, the strawman design used a 64-bit counter for each basic block. Supporting up to 2^{64} bits is important, but in practice, bit arrays are not *too* large. Therefore, for each 2^{32} bits (an *upper block*), we store a 64-bit counter to store the number of one bits to the left of that upper block.

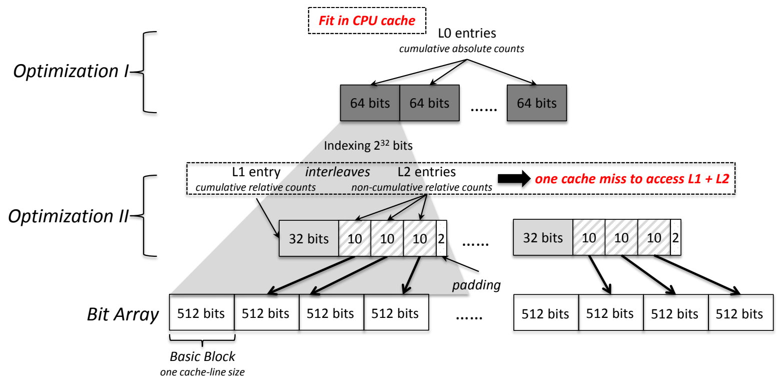


Fig. 4: Our Rank Structure

These 64-bit counters create a new index layer, called the first layer (L0) index. When answering a query for rank(x), we examine this index to find the number of one bits preceding the upper block in which x is located, and look up the underlying structure to find out the number of one bits preceding x within that upper block.

Accessing this additional index does not significantly affect performance for two reasons: First, the L0 index is small enough to fit in fast cache memory. It contains only 64 bits for each 2^{32} bits in the original array. For a bit array of 16 billion bits (2GB), it requires only 128 bytes. Second, the lookup in this index is independent of the lookup in the second-layer index, so these operations can be issued in parallel. This additional layer of index confers an important space advantage: The underlying indexes now need only support 2^{32} bits, so we can represent each 512-bit basic block using only a 32-bit counter. This design results in a rank structure with about 6.25% extra space.

Optimization II: Interleaved L1/L2. We can further improve the space overhead by adding an additional layer to the index. Recall our architectural insight that the overall performance is strongly determined by the number of cache misses, which implies that if no more cache misses are introduced, *slightly* more computation will have minimal performance impact. According to this idea, we designed a two-layer index to support rank queries for 2^{32} bit ranges. For each *four* consecutive basic blocks (a *lower block*, containing 2048 bits), we use a 32-bit counter to store the number of one bits preceding that lower block. These counters make up the second layer (L1) index. Underneath, for each lower block, we use three 10-bit counters, each storing the popcount value for one of the first three basic blocks within that lower block. These 10-bit counters make up the third layer (L2) index. To look up a block, it is necessary to *sum* the appropriate third-layer index entries, but because there are only three such entries, the cost of this linear operation is low.

To avoid causing extra cache misses, we leverage the technique of Vigna [10]: storing the L1 and L2 index entries in an *interleaved form*. Each L1 index entry is followed by its L2 index entries. Since the total size of an L1 index entry and its L2 index entries is 62 bits, which fits in one cache line, this design guarantees that by paying exactly one cache miss, we are able to fetch all the necessary data to answer a rank query. (We pad the structure by two bits to ensure that it is both cache and word aligned.) Even though

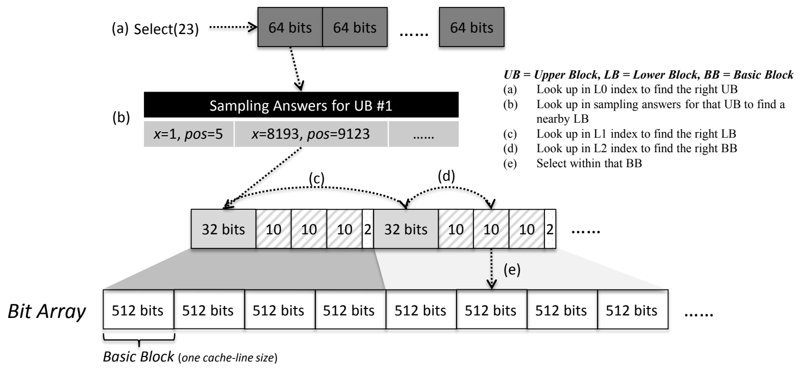

(a) Select(23)
64 bits
64 bits
......
64 bits
UB = Upper Block, LB = Lower Block, BB = Basic Block
(a) Look up in L0 index to find the right UB
(b) Look up in sampling answers for that UB to find a nearby LB
(c) Look up in L1 index to find the right LB
(d) Look up in L2 index to find the right BB
(e) Select within that BB
Sampling Answers for UB #1
(b)
x=1, pos=5
x=8193, pos=9123
......
(c)
(d)
32 bits
10
10
10
2
32 bits
10
10
10
2
......
(e)
Bit Array
512 bits
......
Basic Block (one cache-line size)

Fig. 5: Process of Answering a Select Query

several additional comparisons and arithmetic operations must be performed, the overall performance is only slightly reduced. Because each 2048 bits of the bit array require 64 bits, the space overhead is 3.125%.

Of note is that each layer of the index uses a different type of count: The first layer uses 64-bit cumulative, absolute counts. The second layer uses cumulative counts relative to the beginning of the upper block, and so fits in 32 bits. The third layer uses non-cumulative, relative counts in order to fit all three of them into less than 32 bits, a design constraint required to ensure that the L1/L2 index entries could always be cache-line aligned. The combination of these three types of counts makes our high-performance, space-efficient rank structure possible.

3.2 Select

`combined sampling` [9] is the highest-performing of the space-efficient variants of select, which uses position-based selection. We therefore focus on it as a target for applying our cache-centric optimization and improvements from rank. Our goal, as with `combined sampling`, is to enable maximal re-use of index space already devoted to rank. In contrast, many prior approaches [1,10] create an entirely separate index to use for position-based selection, which requires considerable extra space. As we show, our rank structure, `poppy`, is a natural match for the combined design, and enables support for larger (up to 2^{64} bits) bit arrays while offering competitive or even better performance.

Basic Block for Select Similar to rank, we first microbenchmark the best in-block select method. The result shows that broadword programming [10] is the best method to select within 64 bits. Because the `popcnt` instruction is the fastest way to popcount 512 bits, we combine these two techniques to select within 512 bits—`popcnt` sequentially through the basic block to find the 64-bit word in which the target is located, and then use broadword select to find the individual bit within the word.

Sampling Answers Like other position-based select structures, we store a sampling of select answers.

Strawman Design. The strawman design is for every L one bits, we store the position of the first one among them, which requires 64 bits. We set L to 8192, as in `combined sampling`. To answer a query for `select`(y), we first examine the sampling answers to find out the position of the $(\lfloor (y-1)/8192 \rfloor \cdot 8192 + 1)$-th one bit. We re-use the L1 index of the rank structure to reach the correct *lower block*, and look up in the L2 index of that lower block to find the correct basic block. Finally, we use the combination of `popcnt` and broadword programming to select within that basic block. In the worst case, such a structure adds about 0.78% extra space.

Optimization: 64 bit L0. The same idea from our rank optimization can be used for select, splitting the index into a 64-bit upper part and 32-bit lower part. We binary search the L0 index of the rank structure to find out the upper block in which the y-th one bit is located. For each upper block, we store a sampling of answers similar to the strawman design, but this time only 32 bits are required to store a position. Then the process of answering a select query is similar to that of strawman design, except that it requires one more look up, as shown in Figure 5. This re-use of the L0 index halves the space overhead, from 0.78% to 0.39%. Because we re-use our `poppy` structure as a building block, we call this select structure `cs-poppy`.

This design is similar to `combined sampling`, with two important differences. First, `cs-poppy` can support up to 2^{64} bits. Second, the rank index allows `cs-poppy` to locate the correct position to within 512 bits, instead of `combined sampling`'s 1024-bit basic block, requiring (potentially) one less cache miss when performing select directly within a basic block. `cs-poppy` thus outperforms `combined sampling` and is performance competitive with the much less space-efficient `simple select`.

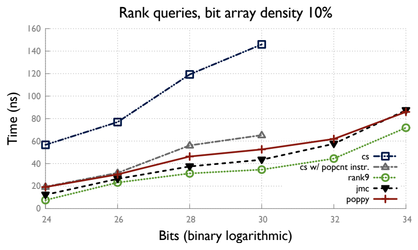
Micro-optimization: Jumping to offset. The sampling index indicates the L1 block containing the sampled bit. Our select performs a small optimization to potentially skip several L1 blocks: Each L1 block can only contain 2048 one bits. Therefore, for `select`(y), it is safe to skip forward by $\frac{y\%8192}{2048}$ L1 entries. This optimization improves select performance by 12.6%, 2.4%, and 0.5% for 2^{30}-entry bit arrays consisting of 90%, 50%, and 10% ones, respectively.

4 Evaluation

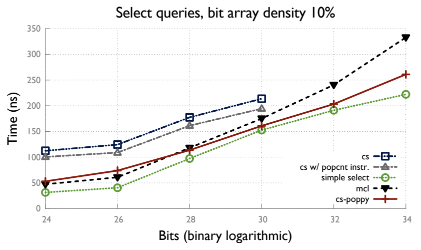
To evaluate our rank & select structures, we performed several experiments on the Intel Core i7-based machine mentioned above. The source code was compiled using `gcc` 4.7.1 with options `-O9`, `-march=native` and `-mpopcnt`. We measure elapsed time using the function `gettimeofday`.

We pre-generate random bit arrays and 1,000,000 test queries before measurement begins. Each test is repeated 10 times. Because the deviation among these runs is small, we report the mean performance. We execute rank & select queries over bit arrays of densities 10%, 50%, and 90%, similar to the experiments by Navarro et al. [9].

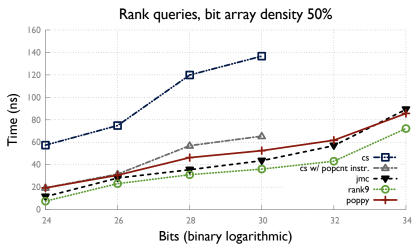
For rank, because our goal is to provide the performance of `rank9` while matching the space overhead of `combined sampling`, we compare `poppy` with these two. We also compare with SDSL [4]'s `rank_support_jmc`, which implements the classical solution [6]. Figure 6 shows the results. For small bit arrays, our rank structure, `poppy` performs slower than `rank9` and `rank_support_jmc`. The performance gap shrinks as the size of bit array increases. When the size is increased to 2^{34}, `poppy`'s



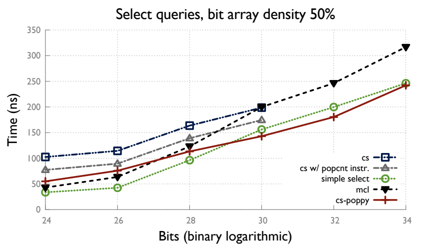
(a)



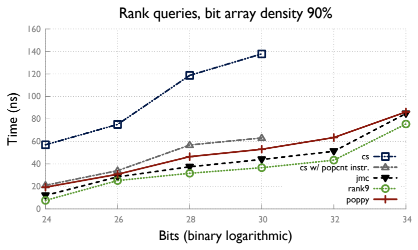
(b)



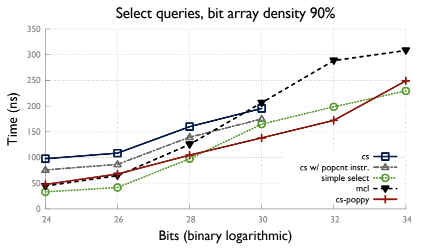
(c)



(d)



(e)



(f)

Fig. 6: Performance of rank & select operations in bit arrays of increasing size

performance is competitive or even better. For small arrays, the index structure fits in cache, and the relative cost of `poppy`'s extra computation adds measureable overhead, but as the index grows to exceed cache, this extra arithmetic is overshadowed by the cost that all schemes must pay to access DRAM. In fact, `poppy` can out-peform the other schemes for index sizes where `poppy`'s smaller index fits in cache and the others do not. On the other hand, `poppy` is substantially faster than the original implementation of `combind sampling` To understand why, we modified the original implementation to use the `popcnt` instruction. `poppy` still outperforms this modified implementation of `combined sampling` by 20%-30%, which we believe is mainly from the new cache-aware index structure design.

For select, we compare `cs-poppy` against `simple select`, `combined sampling`, `combined sampling` using `popcnt`, and SDSL's `select_support_mcl`, which is an implementation of Clark's structure [1] enhanced by broardword programming.

As shown in Figure 6 (b), (d), and (f), `cs-poppy` performs similarly or better than `simple select` and `select_support_mcl`, and always outperforms `combined sampling` and its variant. This result matches our analysis that `combined sampling` may require one cache miss more than `cs-poppy`, because its basic block occupies two cache lines (1024 bits).

5 Conclusion

In this paper, we overview several representative rank & select data structures and summarize common design frameworks for such structures. Then, we present our design recipe for each component, motivated both algorithmic and computer architecture considerations. Following our design recipe, we build space-efficient, high-performance rank & select structures on a commodity machine which support up to 2^{64} bits. The resulting `poppy` rank structure offers performance competitive to the state of the art while adding only 3% extra space; building upon it, `cs-poppy` offers similar or even better select performance than the best alternative position-based select, while adding only 0.39% extra space.

Acknowledgments

We gratefully acknowledge: the authors of `combined sampling` for providing their source code for comparison; Bin Fan and the SEA reviewers for their feedback and suggestions; and Google, the Intel Science and Technology Center for Cloud Computing, and the National Science Foundation under award CCF-0964474 for their financial support of this research.

References

1. David Richard Clark. *Compact pat trees*. PhD thesis, Waterloo, Ont., Canada, Canada, 1998.
2. Peter Elias. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM*, 21(2):246–260, April 1974.
3. R. M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, Project MAC*, 1971.
4. Simon Gog. https://github.com/simongog/sdsl.
5. 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 (WEA05)*, pages 27–38, 2005.
6. G. Jacobson. Space-efficient static trees and graphs. In *Proc. Symposium on Foundations of Computer Science*, SFCS '89, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.
7. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.
8. Susana Ladra, Oscar Pedreira, Jose Duato, and Nieves R. Brisaboa. Exploiting SIMD instructions in current processors to improve classical string algorithms. In *Proc. East European conference on Advances in Databases and Information Systems*, ADBIS'12, pages 254–267, 2012.
9. Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *SEA*, pages 295–306, 2012.
10. Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. International Conference on Experimental Algorithms*, WEA'08, pages 154–168, 2008.