Elias-Fano Compression for Space-Efficient Rank and Select Structures

Lannie Dalton Hough
Department of Computer Science, University of Maryland, College Park, MD, USA

Abhinav Bhatele
Department of Computer Science, University of Maryland, College Park, MD, USA

Abstract

Bit vectors are an important component in many data structures. Such data structures are used in a variety of applications and domains including databases, search engines, and computational biology. Many use cases depend on being able to perform *rank* and/or *select* queries on the bit vector. No existing rank and select structure enabling these queries is most efficient both for space and for time; there is a tradeoff between the two. In practice, the smallest rank and select data structures, `cs-poppy` and `pasta-flat`, impose a space overhead of 3.51%, or 3.125% if only rank needs to be supported. In this paper, we present a new data structure, `orzo`, which reduces the overhead of the rank component by a further 26.5%. We preserve desirable cache-centric design decisions made in prior work, which allows us to minimize the performance penalty of creating a smaller data structure.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Data compression

Keywords and phrases rank and select, cache-aware, succinct data structures, bit vector

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.23

Supplementary Material *Software (Source Code)*: `https://github.com/hpcgroup/orzo` archived at `swh:1:dir:e68f429dc6d8fb21e284b1052e82305b12926410`

Funding This material is based upon work supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, through solicitation DE-FOA-0003264, "Advancements in Artificial Intelligence for Science", under Award Number DE-SC0025598.

Acknowledgements The authors acknowledge the University of Maryland supercomputing resources made available for conducting the research reported in this paper. We would also like to thank Dr. Rob Patro for sharing valuable insights into this area of research.

1 Introduction

Rank and select data structures support *rank* and *select* queries on bit vectors. Informally, a rank query counts the number of 1s or 0s in a bit vector, up to the i-th position, while a select query finds the position of the i-th 1 or 0. Efficient support for these data structures is important, due to their role in the design of a multitude of more sophisticated data structures. Examples include wavelet trees [7], as well as compressed suffix arrays and trees [8], commonly used for text indexing and in search engines. They have also been applied as a component in approximate membership query filters [15], which are utilized in numerous domains, including databases and computational biology. In sequences of monotonically non-decreasing integers compressed by the Elias-Fano encoding [2][3], select queries are necessary for efficient random access. Such sequences are found in inverted indexes, also commonly used in search engines [21].

In domains where rank and select data structures are utilized, memory is often at a premium. As such, in addition to query runtime, it is important to consider memory utilization when deciding which rank/select structure to use. One of the major challenges in the design of rank and select data structures is that reducing query time and reducing the memory footprint of the data structure are often at odds. Frequently, tradeoffs must be made in order to improve one or the other.

Performing rank and select efficiently requires an auxiliary data structure, otherwise a linear scan of the bit vector is required. The naïve approach of precomputing the results for all possible queries and storing these results in a fixed-width integer array is clearly very space-inefficient, with the space requirements of the new array dominating those of the original bit vector. *Succinct* data structures can store this auxiliary information much more compactly. Succinct data structures are those that require an amount of space close to the information-theoretic lower bound. If M is the information-theoretical optimal number of bits needed to store some data, a succinct representation takes $M + o(M)$ bits of space.

While prior work has explored the design of *theoretically* optimal succinct rank and select data structures [19, 16], these are generally not practical, either due to high constant factors or a lack of regard for the realities of hardware architectures. For example, it has been observed and is now well-understood that cache-misses play a significant role in determining the runtime of rank and select queries [6]. As such, minimizing potential cache misses is an important design goal, and a challenge in the design of practical rank and select structures.

Practical rank and select structures do exist, and come with a wide variety of space and time tradeoffs. Because rank and select structures lend themselves particularly well to very large data sets and applications where memory usage is a bottleneck, in this work we focus primarily on the smallest of these structures, modifying their design to use even less memory. We work to preserve principled cache-centric design decisions made in prior work. The main contribution of this paper is the rank and select data structure, `orzo`, which saves space by making use of the aforementioned Elias-Fano encoding [2][3]. While this encoding is typically applied to very long sequences of integers, we instead leverage it at a very small "micro" scale, compressing many short spans of integers.

2 Background and Related Work

Informally, a rank query counts the number of 1s or 0s in a bit vector, up to the i-th position, while a select query finds the position of the i-th 1 or 0. Formally,[1] for a 0-indexed bit vector B of length n:

$$rank_\alpha(i) = \sum_{j \in [0, i-1)} \mathbb{I}(B[j] = \alpha), i \in [1, n], \alpha \in \{0, 1\} \tag{1}$$

$$select_\alpha(i) = min(\{j \in [1, n] : rank_\alpha(j) = i\}) - 1, \alpha \in \{0, 1\} \tag{2}$$

For example, on the 0-indexed 10-bit vector `1011001101`, $rank_1(4) = 3$ and $select_1(4) = 6$. Mentions of rank and select in this paper are assumed to refer to $rank_1$ and $select_1$; generalizing the techniques described to the $\alpha = 0$ variants is straightforward.

[1] The literature varies on whether or not rank is defined as the count of bits up to, or *up to and including* the i-th bit. We use the former definition, as do the implementations against which we compare our approach.

We first discuss the design of rank-only data structures. Practical rank structures often follow a common general design pattern:

- Starting from a *basic block* of size ≥ 2 bits, conceptualize a hierarchy of increasingly large sub-blocks of the bit vector. The largest block size is referred to as a *super block* (or *upper block*).
- Design a multi-layer indexing scheme to provide cumulative counts of one bits across different levels of blocks. The top layer, L0, has one index for each super block, and provides an absolute cumulative count of one bits up to the start of the corresponding block. The next layer, L1, provides cumulative counts up to the start of the next smallest block size, *within a super block*, and so on. Two or three layers typically works well in practice, with the middle blocks in a three-layer scheme often referred to as a *lower blocks*.
- Observe that as block size decreases, corresponding indices need fewer bits to store the cumulative sums within the block.
- When possible, further optimize the design of the data structure around minimizing cache misses. González [6] provides an extensive analysis of the effect that cache misses have on rank and select performance, concluding that they become an increasingly dominant part of the total query time as the size of the bit vector increases.

The size chosen for the basic block has a significant impact on the overall size of the rank and select data structure. The size of the basic block is inversely proportional to the size of the structure, but increasing the size of the basic block also results in longer query times.

2.1 `poppy` and `pasta`

Zhou's `poppy` [20] data structure is a good example of a very small practical rank structure. It uses a basic block spanning 512 bits, a lower block spanning four basic blocks (2048 bits), and super blocks spanning 2^{32} bits, so it possesses a three-layer index. L0 indices are 64 bits wide and count across super blocks, enabling `poppy` to support bit vectors of up to 2^{64} bits. L1 indices count across lower blocks (within super blocks) and are 32 bits wide.

`poppy` diverges from the general design above by using non-cumulative L2 indices to store counts within a lower block. Three L2 indices are used per lower block, storing the population count of the first three basic blocks. Each L2 index needing to store a maximum value of 512 means that they need to be at least ten bits wide. Originally proposed by Geary [5], `poppy` then interleaves the L1 and L2 indices, guaranteeing that it will take no more than one cache miss to access both. Two bits of padding ensure that the resulting L1-L2 index is word-aligned.

This scheme results in a space overhead of only 3.125% relative to the original bit vector, and enables `poppy` to be only slightly slower for rank queries than other structures, such as `rank9` [17], which uses a two-layer index and imposes 25% additional space. `combined sampling` [13] is a one-layer solution that also imposes a space overhead of 3.125%, but achieves this by having a larger basic block of 1024 bits (slowing queries) and by only supporting up to 2^{32} bits.

Kurpicz's `pasta-flat` [11] (see Figure 1) slightly improves upon `poppy`'s performance by doubling the size of the L1-L2 index to 128 bits and storing seven *cumulative* L2 indices of 12 bits each, supporting a lower block size of 4096 bits (eight basic blocks).[2] This leaves $128 - (12 \times 7) = 44$ bits for the L1 index, so super blocks can cover 2^{44} bits. Because both

[2] Though we do not find these performance improvements to be consistent across all input sizes and query types, see Figure 3.

the L1-L2 index and the lower block double in size, the space overhead remains the same as `poppy` at 3.125% (each L1-L2 index covers twice as many basic blocks, but is also twice the width). The size of the L0 layer is negligible in both structures and is optional if the bit vector is less than the size of a super block. `pasta-flat` is visualized in Figure 1.

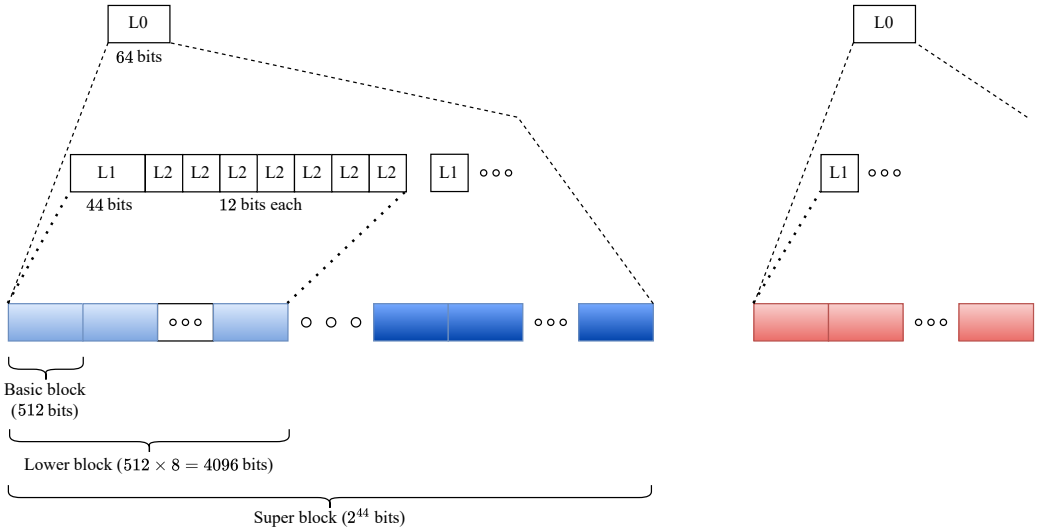

Figure 1 Visualization of `pasta-flat`. The shaded cells represent the original bit vector, while the cells labeled L0, L1, and L2 represent the indices of the `pasta-flat` data structure. One L1 index and seven L2 indices are interleaved in a 128-bit span.

Rank operations

In `poppy`, $rank(i)$ queries are answered by identifying which super and lower blocks contain i, summing their respective L0 and L1 indices, and scanning L2 indices (because they're non-cumulative) to find the basic block containing the i in question. Within a basic block, no information about the ranks of individual bits is stored, so we must perform a scan. Luckily, this can be done very quickly with population count (`POPCNT`) instructions. This instruction returns the number of one bits in a 64-bit word. We therefore need up to $\frac{512}{64} = 8$ popcounts to scan the basic block. This is the fastest method to compute rank within a basic block. `pasta-flat` uses essentially the same procedure, but with cumulative L2 indices it can random access the correct basic block, removing the need for `poppy`'s short L2 scan.

Select operations

For selection there are two methods that are generally employed:

1. In *rank-based selection*, a rank structure like those described above is used to help identify the basic block where the i-th one bit is located. However, this must be done by searching (either a linear scan or binary search), because we cannot know in advance in which basic block the i-th one bit will be. This is in contrast to computing rank, where the i-th bit (one or zero) is always in the same location.
2. In *position-based selection*, select answers for every k ones are sampled, and for answering $select(i)$ we find the largest value of j where $jk \leq i$. The samples allow us to reach a position very close to the correct basic block, which we can then find by scanning.

After the correct basic block is found, in-block selection is performed. A fast way to do this proceeds similarly to rank queries, by popcounting words until the word in which the target is located is identified. In-word selection is then performed. Broadword programming [9] can achieve this quickly in an architecture independent way, while specialized instructions on x86 processors can achieve it even faster [14].

`combined sampling` and `cs-poppy` (the select-enabled variant of rank-only `poppy`) both set the sample rate k to 8192 bits, and make use of their existing rank structures to help answer select queries faster, by combining both rank-based and position-based selection. `cs-poppy` divides these samples into buckets: one for each super block. This facilitates using only 2^{32} bits per sample, which `combined sampling` also uses because it only supports up to 2^{32} bits. The space overhead for both is therefore the same, at 0.39% in addition to the space already used for rank (3.125%, for a total space overhead of 3.51%), relative to the size of the original bit vector.

To answer select queries, `cs-poppy` first identifies the super block containing the answer by scanning the L0 layer. It then identifies a lower block near the answer by using the stored select samples, and identifies the correct lower block and basic block by scanning L1 and L2 indices. In-block selection is performed as described above. `pasta-flat` uses the same design and approach, but also introduces a technique for using SIMD instructions to more quickly identify the correct L2 entry in an L1-L2 index.

2.2 Other Rank/Select Structures

`SPIDER` [12] is another rank and select data structure worth noting, as its space requirements are near that of `poppy` and `pasta-flat`. Inspired by learned data structures [4][10], `SPIDER` uses predictions to speed up select queries, and improves rank performance by interleaving the rank data structure with the original bit vector (for cases where this is undesirable, a non-interleaved variant is proposed where the rank-only component is identical to `pasta-flat`). However, it achieves speedups by using more space than `pasta-flat`, with a total space overhead of 3.82%, or 3.33% to support only rank. It offers good space/time tradeoffs, but in the opposite direction that we are pursuing in this paper.

2.3 Elias-Fano Encoding

Elias-Fano encoding is an encoding scheme which facilitates the compression of a monotonically non-decreasing list of integers [2][3]. In the Elias-Fano encoding, the number of bits required to represent n integers from the universe $[0, u]$ is:

$$2n + n \left\lceil \log_2 \frac{u}{n} \right\rceil \tag{3}$$

The first step in the encoding process is to split each element into its upper $\lceil \log_2 n \rceil$ bits and lower $\lceil \log_2 \frac{u}{n} \rceil$ bits. The lower bits are concatenated to form a bit vector, EF_l, and stored as-is. The upper bits are transformed into a unary representation, a bit vector EF_u of size $2n$ bits. Buckets are created for each possible distinct value $\leq u$ that is representable by the $\lceil \log_2 n \rceil$ upper bits; these buckets store counters for values that are actually represented in the original list. EF_u is then constructed by creating sequences of one bits for the count of each bucket, followed by a zero bit to indicate the end of the bucket. A single bit is set to 1 for each of n elements, and a 0 is used as a stop bit for each bucket. The full representation is the concatenation of the upper and lower bit vectors.

To access an element i in the Elias-Fano representation, the lower bits of the element can be easily extracted from the lower bit vector by computing, and then jumping to, the correct offset. The upper bits for the element must be reconstructed, which is straightforwardly achieved by computing $select_1(i) - i$, essentially a count of the number of buckets passed in reaching the 1 bit of the element in question. The necessity of computing $select_1(i)$ for random access explains why data structures that enable efficient select operations are often used in conjunction with this representation.

3 A New Approach for Saving Space

`cs-poppy` and `pasta-flat` stand out as being the smallest practical rank and select data structures, and their design is extremely effective for minimizing cache-misses and achieving performance near that of much larger structures. For these reasons, we use them as a foundation from which we may consider various optimizations for space savings. There are two strategies by which we can approach reducing space requirements further:

1. Increase the size of the basic block. As mentioned previously, the size of the basic block is inversely proportional to the size of the entire rank structure.
2. Decrease the size of one of the layers. This could be achieved either by reducing the size of individual indices within a given layer, or by reducing the need for as many indices.

The first strategy is straightforward – we do not implement it in this work. The choice of a 512-wide basic block, as used by `cs-poppy` and `pasta-flat`, is empirically well-grounded. Zhou observes that for < 512 bits, doubling the size of the basic block does not double the time it takes to popcount a block, whereas doubling beyond 512 does [20]. As the size of a cache line on modern processors is 64 bytes, we will never incur more than one cache miss in popcounting a basic block of 512 bits, and we allow more memory bandwidth to be free for other operations (assuming that the basic blocks are aligned to cache line boundaries). Although we do not experiment with a larger basic block, we do highlight that the design of `orzo` as described below should generalize to a hypothetical variant with a larger basic block.

The second strategy has different implications when applied to different layers in the rank structure. For example, decreasing the size of the L0 layer is likely to be far less effective in reducing the overall space requirements than reducing the size of the lower layers. Consider the space requirements in bytes of a rank data structure for an N-bit vector with super blocks of size SB_{size} bits, lower blocks of size LB_{size} bits, L0 indices of $L0_{\text{bytes}}$, and L1-L2 indices of $L1L2_{\text{bytes}}$:

$$L0_{\text{bytes}} \times \left\lceil \frac{N}{SB_{\text{size}}} \right\rceil + L1L2_{\text{bytes}} \times \left\lceil \frac{N}{LB_{\text{size}}} \right\rceil \tag{4}$$

For $SB_{\text{size}} \gg LB_{\text{size}}$ the right side of the sum is dominant. With $SB_{\text{size}} = 2^{44}$, $LB_{\text{size}} = 4096$, 64-bit L0 indices, and 128-bit L1-L2 indices in `pasta-flat`, this amounts to well under 0.01% of the total rank structure space, a negligible amount. If all else is kept equal, but the L1 index is reduced to be 20 bits wide (therefore supporting $SB_{\text{size}} = 2^{20}$), the larger L0 layer still consumes only 0.19% of the full size of the rank structure.

Layers which consume more space are also more likely to be responsible for cache misses, as a smaller percentage of the layer is able to fit in various hardware caches. For example, the L0 layers in `cs-poppy` and `pasta-flat` can be cache-resident for even large bit vectors.

With this in mind, the L1-L2 layer is the clear choice to target for space saving optimizations. As previously described, `cs-pasta` and `pasta-flat` interleave an L1 index and several L2 indices into a single L1-L2 index. We want to avoid any approaches that involve splitting

the indices, as an important goal in the design of `orzo` is to introduce as few additional cache misses as possible. *At most*, both of these structures will incur three cache misses in performing a rank query: one to access the L0 index, one to access the L1-L2 index, and one to access the underlying bit vector.

3.1 Trading L1 bits for L2 bits

We observe that in `pasta-flat`, the L1 index is quite wide, and that we might be able to profitably shrink it in order to pack more L2 entries in the L1-L2 index. The size of the lower block can then be increased by 512 for each L2 entry added, and covering more basic blocks without increasing the size of an L1-L2 index will result in space savings. The L0 layer will increase in size, but this increase remains negligible as long as $\text{L2}_{width} \ll \text{L1}_{width}$. `pasta-flat` has seven L2 indices per L1-L2 index, which allows it to have lower blocks that are 4096 bits wide (8 basic blocks). If we add another L2 index, we can cover 4608 bits in a lower block, but we can no longer get away with using only 12 bits per L2 index. This is because previously the maximum number an L2 index needed to represent was $7 \times 512 = 3584$, which can be handled by 12 bits, but 13 bits are required to represent $8 \times 512 = 4096$. As $128 - (13 \times 8) = 24$, we have 24 bits left for the L1 index, not enough space to pack another L2 entry in the L1-L2 index. We refer to this hypothetical variant as `orzo-full`. It is "full" in the sense that it packs as many 13-bit L2 indices as possible in 128 bits, while leaving space for the L1 index.

However, we *can* pack in another L2 index if we represent each L2 index in a different number of bits, depending on the maximum value each L2 needs to store. For example, the first L2 index only needs to store values up to 512, and so can be represented in 10 bits. The second needs to store up to 1024, and so can be represented in 11 bits. With this scheme, we can pack up to nine L2 indices, at the cost of slightly complicating L2 access and shrinking the L1 index to 22 bits. We refer to this variant as `orzo-mixed`.

3.2 Orzo: Elias-Fano Encoded L2 Indices

As it turns out though, we can do better than `orzo-mixed`. L2 indices in `pasta-flat` represent a cumulative count of one bits within the basic blocks in a lower block. As such, they are a candidate to be represented in the Elias-Fano encoding, which can represent monotonically non-decreasing sequences of integers. This is the core observation that facilitates our new approach, which we call `orzo`. The Elias-Fano encoding is, from an information-theoretic perspective, extremely close to the optimal representation for a sequence of monotonically non-decreasing integers [2][1][18], so we should expect this to be a good method of compressing the L2 indices, and indeed it is.

Recall that the space requirements of the Elias-Fano representation depend on the number of elements we want to store, n, and the maximum value that we want to store. If we are attempting to store ten L2 indices in our L1-L2 index, $n = 10$. The maximum number we will need to store is $u = 5120 = (512 * 10)$, the maximum count of ones in ten basic blocks. Putting this together with Equation 2.3, we can calculate the number of bits we need for storing the L2 indices of one lower level block:

$$2 \times 10 + 10 \left\lceil \log_2 \frac{5120}{10} \right\rceil = 110 \tag{5}$$

The L1-L2 index is 128 bits wide, so this leaves us with $128 - 110 = 18$ bits remaining for our L1 index, which is more than enough for the L0 layer to remain extremely small, see Equation 4. In place of distinct L2 indices in our L1-L2 index, we store the concatenated

upper and lower bit vectors of the Elias-Fano representation, EF_u and EF_l respectively. We store EF_u in the lower bits of our L1-L2 index, followed by EF_l, followed by the L1 index, although the order of these is not significant; they could be swapped around and only small changes to the rank and select logic would need to be made. A visualization of `orzo` can be seen in Figure 2.

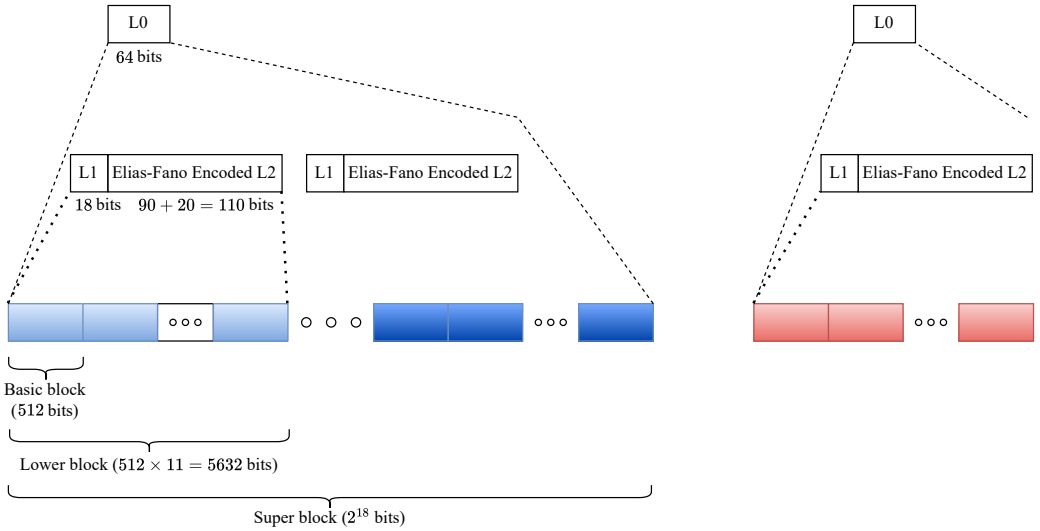

Figure 2 Our rank structure, `orzo`. The overall structure is similar to `pasta-flat` in Figure 1, with the L2 indices compressed by the Elias-Fano encoding. More space in the interleaved L1-L2 index is reserved for L2 indices than in `pasta-flat`, while the L1 index is smaller. As a result, the lower blocks cover a longer span of the original bit vector, while the span of the super blocks is shorter.

With ten L2 indices, our lower block size is $11 \times 512 = 5632$. Because 2^{18} is not evenly divisible by our lower block size of 5632, we opt to make our super block size 259072 instead of exactly 2^{18}; this ensures that a lower block does not span a super block boundary. `orzo`'s space overhead in bytes for a bit vector of N elements is therefore:

$$8\left\lceil \frac{N}{259072} \right\rceil + 16\left\lceil \frac{N}{5632} \right\rceil \tag{6}$$

This is an overhead of 2.297% for rank queries, or ~26.49% less space than the 3.125% `cs-poppy` and `pasta-flat` require for their rank-only components. It is also an improvement over our more naïve `orzo-full` and `orzo-mixed` structures, see Table 1.

Table 1 Space savings in `orzo` versus `pasta-flat`.

Approach	L2 Indices	Space Savings
`pasta-flat`	7	0.0%
`orzo-full`	8	11.1%
`orzo-mixed`	9	19.95%
`orzo`	10	**26.49%**

From a query performance perspective, there is one downside in regards to decreasing the width of the L1 index. While relative to the size of the full rank structure the L0 layer is still extremely small, the input size for which the L0 layer may remain cache-resident decreases. While for `pasta-flat` the L0 layer is not even necessary for input sizes $< 2^{44}$ bits, in `orzo` at 2^{30} bits the L0 layer reaches the size of the level 1 data cache (32 KiB) in one of the processors we use for testing (see Table 2 in Section 6), and reaches the level 2 cache (512 KiB) at 2^{34} bits. Although this is not ideal, it is a necessary tradeoff to obtain the space savings that we do. Introducing the potential for more cache misses in the L0 layer is not nearly as bad as sacrificing the cache properties of the L1-L2 layer. Indeed, we actually slightly *minimize* the potential for cache misses in the L1-L2 layer, as queries on almost all random indices i and j will access L1-L2 indices that are closer together, as a result of our larger lower blocks representing more of the underlying bit vector.

4 Orzo: Rank Queries

Compared to how rank queries are handled in `pasta-flat`, `orzo` introduces only one additional complication: the need to decode an L2 index. The L1 index is extracted in essentially the same way as `pasta-flat` (only the magnitude of the bit shift required changes), as is the L0 index (assuming the L0 index is necessary).

For decoding the L2 indices, the lower part of the Elias-Fano representation is easily extracted with a bit shift followed by a mask, in the same manner that full L2 indices are extracted from `pasta-flat`. The upper bits must be reconstructed. As discussed in Section 2.3, this requires computing $select_1(i) - i$ on the upper Elias-Fano bit vector. At only twenty bits wide, our upper bit vector fits within a machine word, so we can leverage fast in-word selection techniques. The best of these on x86-64 architectures [14] leverages two special instructions: `PDEP` and `TZCNT`. `PDEP` transfers bits from the first operand into a destination based on the mask in a second operand. `TZCNT` counts the number of trailing zero bits in the source and returns the count. For non-x86 architectures, broadword programming [9] provides a relatively efficient way to select within a word. The reconstructed upper bits are concatenated with the lower bits to construct the full L2 index. After decoding the L2 index and adding it to the L1 and L0 indices, the final step is to determine the number of ones present in the basic block, which we can do very quickly with a scan and popcount instructions (maximum of eight).

5 Orzo: Select Queries

In order to speed up select operations, `orzo` samples the position of every 8192nd one bit, in the same fashion as `cs-poppy` and `pasta-flat`. This incurs a maximum additional space overhead of 0.39% compared to the rank-only versions of these structures. Unfortunately, `orzo`'s design complicates select queries more than it does rank queries.

Recall that `cs-poppy` divides select samples into buckets, with one bucket per super block. For $select(y)$, the L0 layer is searched in order to find the super block in which the y-th one bit is present.[3] However, because of its narrower L1 index, `orzo` may have many more L0 entries; searching these would be costly; a scan would have to examine many elements, while a binary search would exhibit poor cache locality and would still visit substantially more entries than the equivalent in `cs-poppy` or `pasta-flat`.

[3] The `cs-poppy` paper describes a binary search, but the available implementation uses a linear scan.

Listing 1 Using x86 intrinsics to select within a word [14].

```
uint64_t pdep_result = _pdep_u64(1ul << i, word);
uint64_t select_result = _tzcnt_u64(pdep_result);
```

To address this, we opt to introduce a *second super block* to `orzo` for select queries, and a corresponding second L0 layer, $L0_{select}$. Instead of covering 259072 ($\approx 2^{18}$) bits, these select super blocks cover 4294895616 bits. This number is chosen instead of exactly 2^{32}, as it is evenly divisible both by our regular super block size and our lower block size. The additional memory incurred by the second super block is negligible.

We can now scan our $L0_{select}$ layer at the same speed as `cs-poppy`, and we can divide our select samples into fewer buckets. We obtain a select sample from our bucket in the same manner as `cs-poppy`, which locates a lower block *near* the target lower block containing the answer to our query. To find the lower block actually containing the answer, we must scan L1 indices.

`orzo` again complicates this by the fact that our L1 indices are only cumulative within our standard super blocks, not our select super blocks. While `cs-poppy` may simply examine the next L1 index and determine if the current block contains the solution to our query, `orzo` first needs to know the rank at the start of the regular super block that the current lower block is within. Luckily, this is simple to retrieve, as knowing what lower block we are in allows us to determine the regular L0 index corresponding to the start of our regular super block. We can now proceed with our scan of L1 indices until we reach the correct lower block. Our scan of L2 indices proceeds in a similar fashion to `cs-poppy`, with the exception that we must first decode the indices. Once the correct basic block is reached, we leverage the technique of using popcount instructions to find the correct word, and for select within a word we use the same `PDEP` and `TZCNT` instructions that we have already described, see Listing 1.

Although `orzo` does introduce some complications for select, its design does have one upside from a query performance perspective that helps offset some of the overhead from the extra bookkeeping. `orzo`'s lower blocks span 5632 bits, in contrast to `pasta-flat`'s 4096 bits. This means that it is sometimes the case that fewer L1 indices must be scanned in order to identify the correct lower block; this is especially true for sparser bit vectors.

6 Experimental Setup

All code for our `orzo` implementation, as well as the code for the implementations we compare against, was written in C++ and compiled with clang version 17.0.6. We performed our experiments on two different x86-64 processors from different vendors, provisioned on compute nodes running Red Hat Enterprise Linux (Version 8.10) through the University of Maryland's Zaratan cluster. See Table 2 for the specifics of each platform. The CPUs supported different features (for example, no AVX512 on the AMD CPU), and we used different compiler flags for each CPU, see Table 3 for details. We always pin our benchmarking processes to CPU 1.

Table 2 CPUs used for our experiments.

CPU	Clock Rate	L1d Cache	L1i Cache	L2 Cache
Intel Xeon Platinum 8468	3.8 GHz	48 KiB	32 KiB	2048 KiB
AMD EPYC 7763	2.23 GHz	32 KiB	32 KiB	512 KiB

Table 3 Compiler flags used in our experiments.

CPU	Compiler Flags
All	`-std=c++23 -O3 -flto -static`
Intel Xeon Platinum 8468	`-mbmi -mbmi2 -mavx512f -mavx512vl -mavx512bw -mavx2`
AMD EPYC 7763	`-march=native`

For our experiments, we mostly replicate the setups used by Zhou and Kurpicz in evaluating `cs-poppy` [20] and `pasta-flat` [11]. We create random bit vectors of various lengths at three different sparsity levels of one bits: 10%, 50%, and 90%. For each input we test ten million random queries, and we run all tests five times with randomized bit vectors. We show the average of these runs in our reported results (the deviation between runs is extremely small).

The following rank and select data structures are evaluated in our experiments:

- `cs-poppy` is Kurpicz's [11] implementation of Zhou's [20] data structure.[4]
- `pasta-flat` is Kurpicz's [11] original version of the `pasta-flat` data structure.
- `orzo` is the rank and select data structure that we describe in Section 3.2, with ten Elias-Fano encoded L2 indices per interleaved L1-L2 index, and an L1 index size of 18 bits.

As the focus of this research is to improve the *space* efficiency of practical rank and select structures, we opt to compare only against structures which incur a space overhead of $\approx 3.51\%$ for rank and select, or $\approx 3.125\%$ for just rank. It is well-known that approaches which use more memory (and possibly make other tradeoffs as well, such as `SPIDER`'s [12] interleaving of the bit vector with the rank/select structure) can achieve better query times. Examples include `pasta-wide`, another rank structure by Kurpicz [11] which incurs an overhead of 10%, and `rank9` by Vigna [17] which incurs an overhead of 25%. We also omit `combined sampling` [13] from our analysis, despite it meeting the specified space requirements. Zhou has demonstrated that it is considerably (2-3x depending on the size of the bit vector) less efficient for rank queries than `cs-poppy`, and slightly less efficient for select queries [20]. Additionally, `combined sampling` only supports bit vectors of up to 2^{32} bits.

7 Experimental Results and Interpretation

For simplicity, in this section, we focus on `orzo`'s query performance relative only to `pasta-flat`. However, as can be seen in the performance plots in Figures 3 and 4, both `pasta-flat` and `cs-poppy` perform very similarly, so our observations apply to both. The performance plots annotate each data point for `orzo` with the percentage increase in query time over `pasta-flat`.

Query performance is generally quite consistent across different input sparsities, especially for rank, across both CPUs that we tested, and for all of the data structures. An exception is that for select, there is a jump in query time going from 50% sparse bit vectors to 90% sparse bit vectors. On the Xeon CPU, going from 50% to 90% sparsity for the 2^{34} input size increases query times from ~156 ns (`orzo`) and ~109 ns (`pasta-flat`) to ~177 ns and ~129 ns. See plots *e* and *f* of Figure 3. This is expected, because select queries must scan more L1 indices with decreasing density in the bit vector.

[4] We opt to use Kurpicz's implementation, as they demonstrate it to be faster than the original implementation, and Kurpicz reports that the original implementation has a bug that prevents select queries from terminating on large bit vectors.

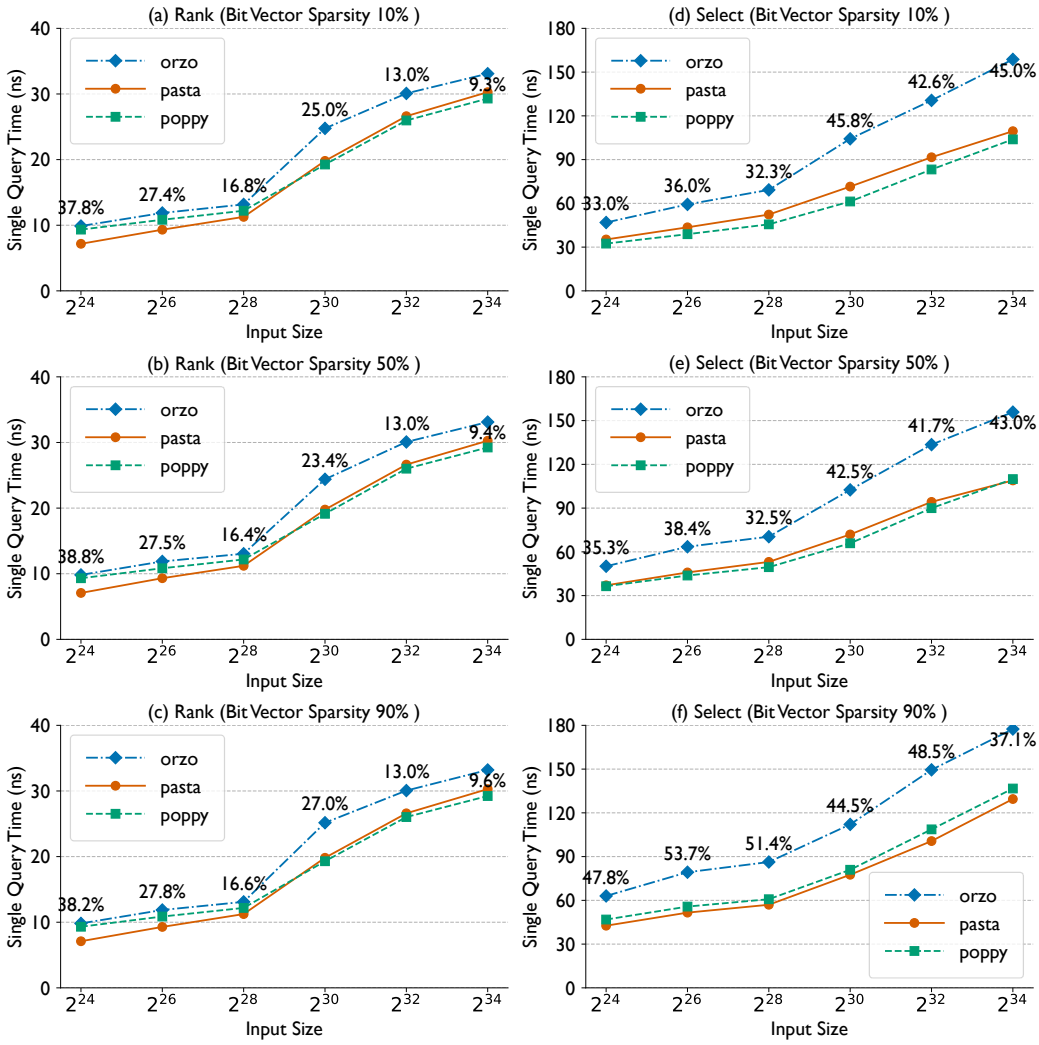

Figure 3 Performance of rank and select operations on Intel Xeon CPU. Data points for `orzo` are annotated with the percentage increase in query runtime compared to `pasta-flat`.

Relative performance between `orzo` and `pasta-flat` is not consistent across CPUs. For example, there is a ~9% gap in query runtime between `orzo` and `pasta-flat` for rank queries on a 2^{34} bit vector using the Intel Xeon CPU. This becomes a ~38% gap on the AMD EPYC. See corresponding *a*, *b*, and *c* plots of Figures 3 and 4. On the other hand, for select queries, the AMD EPYC ends up being the platform with less of a gap in query runtime at large input sizes.

While `cs-poppy` and `pasta-flat` are closer in performance to each other than `orzo` is to either, neither is consistently faster, with the winner varying across different query types, sparsities, and input sizes. This is in contrast to prior comparisons between these two data structures [11], which had `pasta-flat` consistently ahead.

The inconsistency across machines, input sizes, and sparsities implies that it is likely worth benchmarking different rank and select data structures on the hardware and input sizes with which they're going to be used. While `orzo` might represent a good space/time tradeoff on one machine and one range of input sizes, this may not hold in other scenarios. The

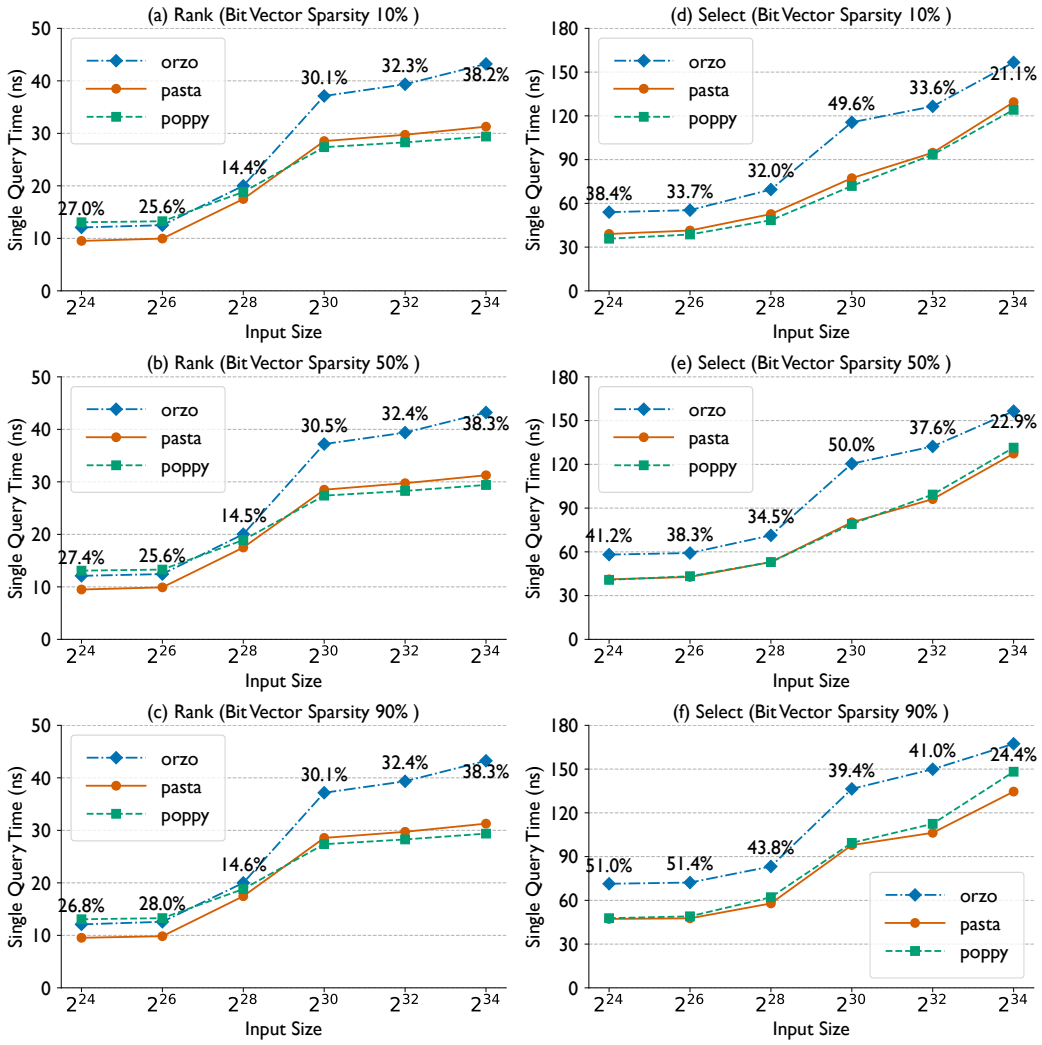

Figure 4 Performance of rank and select operations on AMD EPYC CPU. Data points for `orzo` are annotated with the percentage increase in query runtime compared to `pasta-flat`.

choice between the equally sized `cs-poppy` and `pasta-flat` structures is also less clear-cut than what prior research indicated, with `cs-poppy` actually having faster queries in many scenarios.

8 Conclusion and Future Work

In this work, we examine several of the most space-efficient rank and select data structures. Using one of these as a foundation, we propose a scheme to achieve further compression by exploiting Elias-Fano encoding. Our data structure, `orzo`, realizes considerable space savings of 26.49% over the rank components of `cs-poppy` and `pasta-flat`, achieving a new space/time tradeoff that is on the Pareto frontier of rank and select solutions. Our space savings do come at the cost of query time, but this is expected. We suggest that our solution is ideal for applications where memory is the primary bottleneck or a hard constraint, but fast rank and select queries are still necessary. As long as the query runtime is not the

bottleneck in an application, the relative increases in query time are unlikely to be felt, due to the extremely small runtime of the queries in absolute terms. We also conjecture that our technique of using Elias-Fano encoding for *micro-scale* compression, as opposed to its typical use in compressing large datasets, might find use in the design and engineering of other data structures.

Future Work

There are several topics we intend to address in future work. The first is evaluating the construction time for various rank and select data structures. To date, most studies focus primarily on query times or space improvements.

Revisiting `cs-poppy`-style delta-encoded L2 indices is also of interest. It is a space-efficient representation, and it is worth researching mitigations to the increasing cost of the L2 scan if significantly more L2 indices are added (as would be necessary if moving to a 128-bit L1-L2 index, for example).

Finally, we wish to implement and evaluate rank and select data structures on alternative architectures, such as GPUs. This could potentially ellucidate new use-cases for these structures, and would also be interesting for how it may affect the design of rank and select structures and algorithms. For example, modern NVIDIA GPUs have a cache line width of 128 bytes, twice as large as most modern CPUs. Prior work suggests the size of the basic block could be profitably doubled in such a scenario [20].

References

1 Peter Elias. On binary representations of monotone sequences. In *Proceedings of the Sixth Princeton Conference on Information Sciences and Systems, Department of Electrical Engineering, Princeton University, Princeton, NJ*, pages 54–57, 1972.

2 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974. `doi:10.1145/321812.321820`.

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

4 Paolo Ferragina and Giorgio Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data: Tutorials from the INNS Big Data and Deep Learning Conference (INNSBDDL2019)*, pages 5–41. Springer, 2020. `doi:10.1007/978-3-030-43883-8_2`.

5 Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. In *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings 15*, pages 159–172. Springer, 2004. `doi:10.1007/978-3-540-27801-6_12`.

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

7 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=644108.644250`.

8 Roberto Grossi and JS Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symp. Theory of Computing (STOC'2000)*, pages 397–406. Citeseer, 2000.

9 Donald E Knuth. *The art of computer programming, volume 4, fascicle 1: Bitwise tricks & techniques; binary decision diagrams.* Addison-Wesley Professional, 2009.

10 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. *CoRR*, abs/1712.01208, 2017. arXiv:1712.01208.

11 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In Diego Arroyuelo and Barbara Poblete, editors, *String Processing and Information Retrieval*, pages 257–272, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-20643-6_19.

12 Matthew D. Laws, Jocelyn Bliven, Kit Conklin, Elyes Laalai, Samuel McCauley, and Zach S. Sturdevant. SPIDER: Improved Succinct Rank and Select Performance. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms (SEA 2024)*, volume 301 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:18, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2024.21.

13 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proceedings of the 11th International Conference on Experimental Algorithms*, SEA'12, pages 295–306, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-30850-5_26.

14 Prashant Pandey, Michael A. Bender, and Rob Johnson. A fast x86 implementation of select. *CoRR*, abs/1706.00990, 2017. arXiv:1706.00990.

15 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 775–787, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3035963.

16 Mihai Patrascu. Succincter. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2008. doi:10.1109/FOCS.2008.83.

17 Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA'08, pages 154–168, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-68552-4_12.

18 Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 83–92, 2013. doi:10.1145/2433396.2433409.

19 Huacheng Yu. Optimal succinct rank data structure via approximate nonnegative tensor decomposition. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 955–966, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313276.3316352.

20 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, pages 151–163, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38527-8_15.

21 Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6–es, July 2006. doi:10.1145/1132956.1132959.