Practical Bit Vectors Supporting Constant Time Rank and Select in Optimal Space

Florian Kurpicz
University of Münster, Germany

Niccolò Rigi-Luperti
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Abstract

Bit vectors with support for fast rank and select are a fundamental building block for compressed data structures. We close a gap between theory and practice by mapping a design space of promising data structures, analyzing it, and experimentally evaluating a promising region. The result are implementations of rank and select data structures for bit vectors with worst-case constant query time, leading practical performance, and a space-overhead reaching below 1 %. For difficult inputs, we are ≈ 8 times faster than the best previous implementations.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Data structures design and analysis

Keywords and phrases succinct data structures, bit vectors, rank and select

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.8

Related Version *Previous Version*: `https://arxiv.org/abs/2509.17819n` [22]

Supplementary Material
Software (Source code): `https://codeberg.org/limnopilos/limnopilos` [20]
Text (Link to source code within Limnopilos): `https://codeberg.org/limnopilos/limnopilos/src/branch/main/crates/bit_vector` [19]

1 Introduction

Succinct data structures represent data in space close to the information-theoretical minimum while still supporting efficient queries. To this end, we frequently need the ability to count (*rank*) and locate (*select*) bits in a binary sequence, or *bit vector*. These operations are simple to define yet surprisingly powerful: Given a length-n bit vector $v[0..n) \in \{0,1\}^n$, we have

$$rank_1(i) = \overbrace{\textstyle\sum_{j=0}^{i-1} v[j]}^{\text{count 1-bits up to position } i} \text{ and } rank_0(i) = i - rank_1(i), \text{ and}$$
$$select_x(i) = \underbrace{\arg\min_j(rank_x(j) = i) - 1}_{\text{locate } i\text{-th } x\text{-bit, i.e., first } x\text{-bit with rank } i} \text{ for } x \in \{0,1\}.$$

Static bit vectors with rank and select support are fundamental building blocks for succinct and other space-efficient data structures. For example, a tree with n nodes can be encoded as a $2n$-bit vector [17] such that, using rank and select, many operations like navigation in the tree can be supported [27]. Bit vectors are also central for supporting space-optimal constant-time range minima [9]. A sorted sequence of n integers from a universe of size u can be represented using $n\lceil 2 + \log \frac{n}{u} \rceil$ bits and then efficiently accessed via rank and select queries on a bit vector [5, 6]. Bit vectors also have applications in full-text indexing [8, 11], computational geometry [4, 7, 24, 25], and data analysis [1] in the form of wavelet trees [16].

From a theoretical perspective, bit vectors are well understood. Rank and select queries can be supported in constant time [3, 17] provided we are willing to spend additional space $\Theta(n \log\log n / \log n)$ [13]. There are practical implementations of *rank* that closely follow the theoretical considerations and nicely align with architectural features like vector instructions [14, 12, 21, 33]. However, there is a large gap between theory and practice when we want to support *select*. The best theoretical constructions [13, 31] do not look promising for a direct implementation. They are underspecified or only achieve low space overhead for superastronomical input sizes. Practical implementations [33] have considerable space overhead (more than needed for *rank*) both in concrete numbers and when extrapolating into asymptotic behavior. They also exhibit non-constant worst-case query time [21, 34]. This suggests that the design space for select data structures has not been systematically explored, leaving potentially better designs undiscovered.

Our Contributions. We present a framework based on *summary trees* and *sample trees* that makes this design space explicit (Section 2). We develop select data structures with constant-time queries that require space $o\left(n\frac{\log\log n}{\log n}\right)$ in addition to the rank data structure with its $\Theta\left(n\frac{\log\log n}{\log n}\right)$ space lower-bound. To highlight the unexplored parts of the design space, we express existing select data structures in its context (Section 3).

Finally, we explore the design space using design choices supported by our theoretical analysis (Section 4). Our experimental evaluation shows that our new data structures outperform the previous state-of-the-art. On the tested range of different input distributions, our implementation provide an interesting space-time trade-off while being at least 8 times faster on difficult inputs (Section 5). Here, a similar performance is only achieved by a data structure that requires 60 times more memory. Additionally, our data structures support rank *and* select queries for both `0`- and `1`-bits, whereas most competitors support only select queries for `1`-bits despite requiring a similar amount of memory.

Preliminaries. Throughout this paper, we assume that all rank and select data structures are built for bit vectors of length n containing m `1`-bits. We use $\log x$ for the base-2 logarithm and $i..j$ is shorthand for the integer range $\{i, \ldots, j\}$. We also use this notation for ranges in arrays, e.g., $v[i..j] = v[i] \ldots v[j]$ and $v[i..j) = v[i] \ldots v[j-1]$.

We analyze algorithms in the random access machine (RAM) model [32]. Here, a wide range of operations can be executed in constant time on $O(\log n)$ bits. For example, $\texttt{popcnt}(v[i..i+w])$ (short for population count) returns the number of `1` bits in $v[i..i+w)$ and can be computed in constant time for $w \in O(\log n)$. In practice, modern hardware amplifies this *word-parallelism* through the availability of vector registers of width up to 512 bits. We also frequently use *bit-packing*, i.e., the most elementary way to store integers in the range $0..U-1$ space-efficiently by representing them as $\lceil \log U \rceil$-bit binary numbers.

2 A Simple Framework for Rank and Select Data Structures

We begin by establishing notation for rank and select data structures, largely following prior work [17, 21, 33, 34]. The surprisingly simple but powerful core of the framework is a hierarchy of implicit contiguous intervals of the underlying bit vector. We call these intervals *blocks*. For these blocks, we store auxiliary information (but no interval boundaries) to enable constant-time rank and select queries.

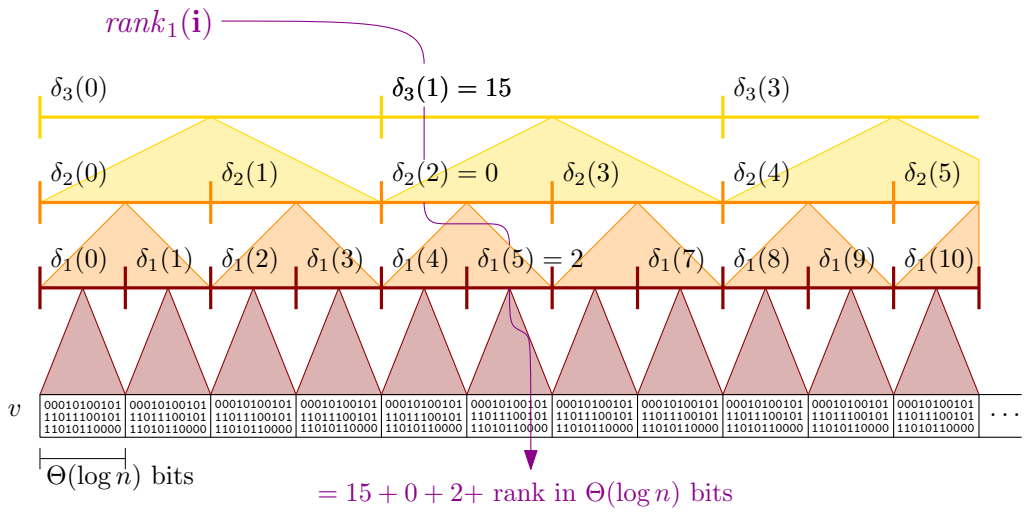

Figure 1 Structure of a 3-level summary tree including the path of a $rank_1(i)$-query.

We use the hierarchy of blocks to define a framework that unifies previous approaches based on *summary trees* (for constant-time rank queries, Section 2.1) and *sample trees* (for constant-time select on top of summary trees, Section 2.2).The framework then enables a parameterized analysis of the space required for constant-time select queries based on the height of the summary and sample trees, and the size of the blocks (Section 2.3).

2.1 Summary Trees: Enabling Constant-Time Rank Queries

We now describe *summary trees*, a constant-time rank data structure for a bit vector $v[0..n) \in \{0,1\}^n$ that is a generalization of a commonly used design, e.g., [17, 31, 21, 33, 34].

Structure. A summary tree consists of at least two levels $\underline{\ell} = O(1)$, numbered $\underline{\ell}..1$, where the root is the $\underline{\ell}$-th level. On level i, the bit vector is partitioned into blocks of size b_i, so that the j-th block covers $v[j \cdot b_i \,..\, (j{+}1) \cdot b_i)$. To obtain a tree-like structure, we require that each block at level $i+1$ is the union of exactly d_{i+1} blocks at level i, giving the recursive relation $b_{i+1} = d_{i+1} \cdot b_i$. For each block j, the summary tree stores auxiliary information accessible via $\delta_i(j)$. The main difference between most constant-time rank data structures is the representation of the data, which we describe in more detail in Section 3.

Auxiliary Data. For constant-time rank queries, we store the following information: On the root level $\underline{\ell}$, we store for each block j the number of 1-bits preceding it: $\delta_{\underline{\ell}}(j) = \texttt{popcnt}(v[0\,..\,j \cdot b_{\underline{\ell}}))$. On every other level $i < \underline{\ell}$, we store for each block j the number of 1-bits within its parent block that precede block j. Let $j' = \lfloor j \cdot b_i / b_{i+1} \rfloor$ be the index of the parent block at level $i+1$. Then, $\delta_i(j) = \texttt{popcnt}(v[j' \cdot b_{i+1} \,..\, j \cdot b_i))$.

Answering a Rank Query. To answer $rank_1(p)$, we descend through the levels from root to base, see Figure 1. At each level i, we determine the block $j_i = \lfloor p/b_i \rfloor$ containing position p and accumulate $\delta_i(j_i)$. At level 0, it remains to count the 1-bits in $v[j_0 \cdot b_0 \,..\, p)$, an interval of at most b_1 bits. A block size of $b_1 = \Theta(\log n)$ allows this count to be computed in constant time using word-parallelism. Thus, we can answer rank queries in constant time.

▶ **Lemma 1.** *Given a bit vector of length n, a summary tree with $b_1 = \Theta(\log n)$ and $d_i = \Theta(\log n / \log\log n)$ for all $i = 2..\underline{\ell}$ with $\underline{\ell} = O(1)$* **(i)** *can be constructed in $O(n/\log n)$ time,* **(ii)** *can answer rank queries in $O(1)$ time, and* **(iii)** *requires $O(n \log\log n / \log n)$ space.*

Proof. To compute a summary tree, we require the number of bits per block on level 1, which we can compute in $O(n/\log n)$ time using word-parallelism. Afterwards, we only have to scan the information of these blocks once per level, i.e., a constant number of times. Hence, the overall construction time is $O(n/\log n)$.

To answer a rank query, we only access the information of one block per level, which we can do in constant time. The block can also be identified in constant time. Since $rank_0(p) = p - rank_1(p)$, we can answer all rank queries in constant time.

We now analyze the space requirements. On level 1, for each block j, we can express $\delta_1(j)$ in $O(\log\log n)$ bits. There are $\Theta(n/\log n)$ such blocks, hence, level 1 requires $O(n \log\log n/\log n)$ bits of space. For all levels $i = 2..\underline{\ell} - 1$, we store relative counts. On each such level, the number of bits increases by the factor d_i. However, the number of blocks decreases by the same factor. Thus the most space is needed for level 1. At the root level, we store absolute counts requiring $O(\log n)$ bits each. The number of blocks at the root level is $O\left(n \frac{(\log\log n)^{\underline{\ell}-2}}{(\log n)^{\underline{\ell}-1}}\right)$. Hence, the space for the root level is $O\left(n \frac{(\log\log n)^{\underline{\ell}-2}}{(\log n)^{\underline{\ell}-2}}\right)$ bits, which is dominated by the space for level 1. ◀

Support for Select Queries. Select queries cannot be answered in constant time on the whole bit vector using just a summary tree. However, we can answer them within any block in constant time, given the correct root-level block. We call blocks at root level *super blocks*.

▶ **Lemma 2.** *Given a super block j that contains the p-th bit, a summary tree as defined in Lemma 1 can answer $select_1(p)$ in $O(1)$ time.*

Proof. The auxiliary data suffices for fast select in a super block. Starting at the root level block containing the p-th bit, whenever we are in a block at level i, we only have to consider one of its $d_i = O(\log n/\log\log n)$ covered blocks at level $i-1$. The maximum count, i.e., the number of 1-bits covered by a block is at most $O((\log n)^{\underline{\ell}}/(\log\log n)^{\underline{\ell}-1})$ and can be represented using $O(\log\log n)$ bits. Thus, all these counts can be represented in $O(\log n)$ bits, allowing us to search through them in $O(1)$ time using word-parallelism [33]. On level 1, a block has size $\Theta(\log n)$, allowing $O(1)$ search time using word-parallelism. ◀

This allows us to identify the child block that contains the searched bit for a select query, but does not address the identification of the initial super block containing the p-th bit.

Smaller Summary Trees. We can also consider summary trees with 0 or 1 level. In both cases, we cannot support constant-time rank queries. A 0-level summary tree stores the positions of all 1-bits. This is sufficient for constant-time select, but cannot achieve succinctness. A 1-level summary tree means that we have blocks of size $\Theta(\log n)$ and only relative information, i.e., the count of 1-bits within each block. Here, we can support constant-time select (but not rank) when we know the super block containing the queried bit.

2.2 Sample Trees: Enabling Constant-Time Select Queries

Super blocks are the blocks at the root level of a summary tree. By Lemma 2, given the super block that contains the p-th 1-bit, we can answer $select_1(p)$ in constant time. To identify this super block, we sample super block indices at increasingly finer resolution in a *sample tree*.

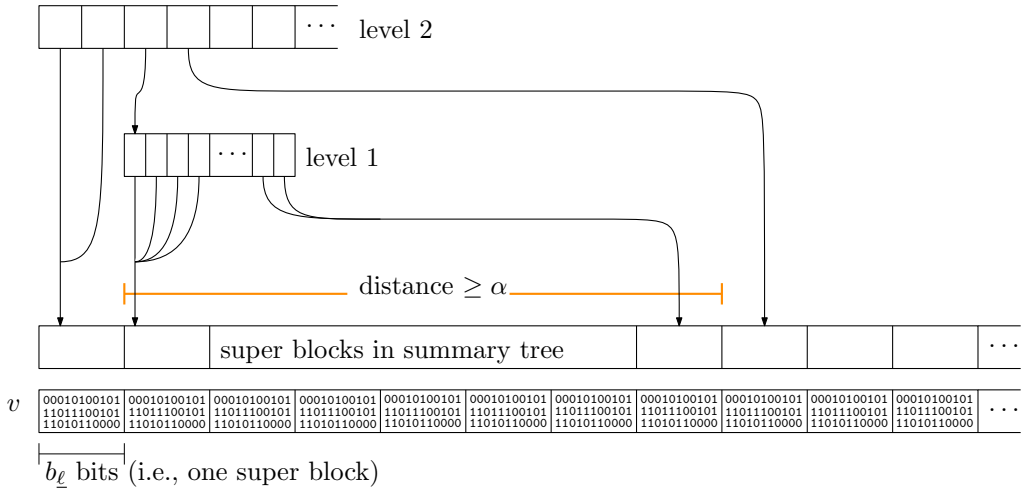

Figure 2 Structure of a 2-level sample tree. When the distance between two super blocks is $\geq \alpha$, every 1-bit is sampled in level 1.

Structure. A sample tree consists of at least two levels $\overline{\ell} = O(1)$, numbered $\overline{\ell}..1$, where level $\overline{\ell}$ is the root. Each level i has a *sample rate* a_i, with $a_{\overline{\ell}} > \cdots > a_2 > a_1 = 1$. On level i, the sample tree stores the super block index of every a_i-th 1-bit. Here, $\sigma_i(j)$ denotes the j-th sample at level i. Two consecutive samples $\sigma_i(j)$ and $\sigma_i(j{+}1)$ on the same level define a *range* of $r = \sigma_i(j{+}1) - \sigma_i(j)$ super blocks that together contain all 1-bits between the two samples. We distinguish two cases based on a threshold $\alpha = O(1)$:

- $r < \alpha$ (*small* range): Search through a constant number of super blocks and answer select query within it in constant time using the summary tree (Lemma 2).
- $r \geq \alpha$ (*large* range): Recurse into the next level, with its finer samples within this range.

On level 1, we have $a_1 = 1$, so the super block can be read off directly, see Figure 2.

▶ **Observation 3.** *There are at most $L = n/(\alpha \cdot b_{\underline{\ell}})$ large ranges, since each large range spans at least α super blocks covering at least $\alpha \cdot b_{\underline{\ell}}$ positions.*

Answering a Select Query. To answer $select_1(p)$, we start at the root level $\overline{\ell}$ and determine the two consecutive samples that enclose the p-th 1-bit, i.e., $j = \lfloor p/a_{\overline{\ell}} \rfloor$. If the range $\sigma_{\overline{\ell}}(j{+}1) - \sigma_{\overline{\ell}}(j)$ is small, we identify the super block directly. Otherwise, we descend to the next level and consider only the samples between the two enclosing samples at the level above, repeating the same case distinction. We do this recursively, until we find a small range. At level 1, the super block index is stored explicitly.

Once the super block is identified, we answer the select query within it in constant time using the summary tree (Lemma 2). Since the sample tree has $\overline{\ell} = O(1)$ levels and each level requires constant time, the overall query time is $O(1)$.

2.3 Space Analysis: Mapping a Design Space for Select Data Structures

Using our framework, we can now identify useful configurations with respect to the number of levels of the summary tree and sample tree. First, we assume that each sample in the sample tree uses one word w of space, i.e., the samples are not compressed.

▶ **Theorem 4.** *For a bit vector of length n, a constant-time rank and select data structure with $\overline{\ell} \in O(1)$ levels of sample tree and $\underline{\ell} \in O(1)$ levels of summary tree requires*

$$O\left(n\frac{(\log\log n)^{(\underline{\ell}-1)(1-1/\overline{\ell})}}{(\log n)^{\underline{\ell}-1-\underline{\ell}/\overline{\ell}}}\right) \text{ bits of space on top of the space needed for the summary tree.}$$

Proof. We first consider the required storage space as a function of various parameters. Let $w = O(\log n)$ denote the number of bits we use to encode a reference to a super block or to a sample on the next level. At the root level, we store $m/a_{\overline{\ell}}$ such references in space $wm/a_{\overline{\ell}}$, one for every $a_{\overline{\ell}}$-th of the bit vector's m 1-bits. In each subsequent level of the sample tree, there are at most $L = n/(\alpha b_{\overline{\ell}})$ large ranges (Observation 3). This takes space wLa_{i+1}/a_i at level i. Using an $\underline{\ell}$-level summary tree with $b_1 = \Theta(\log n)$ and degrees $d_i = \Theta(\log n/\log\log n)$ leads to superblock size $B = b_{\underline{\ell}} = (d_i)^{\underline{\ell}-1} \cdot b_1 = \Omega((\log n)^{\underline{\ell}}/(\log\log n)^{\underline{\ell}-1})$. Note that the base case $a_1 = 1$ as required for a constant time query. For the sampling rates in the higher levels $i \geq 2$ we choose a uniform hierarchy $a_i = a_2^{i-1}$. This leaves a_2 as the only free parameter, which we now choose space-optimally. We get space $wm/(a_2)^{\overline{\ell}-1}$ at the root level and space $wLa_2 = wna_2/(\alpha B)$ at all lower levels. We now set a_2 such that both values are the same. Solving $\frac{wm}{a_2^{\overline{\ell}-1}} = \frac{wna_2}{\alpha B}$ yields $a_2 = \sqrt[\overline{\ell}]{\frac{m}{n}\alpha B}$. Thus, total space consumption is $\overline{\ell}wm\left(\frac{n}{m\alpha B}\right)^{\frac{\overline{\ell}-1}{\overline{\ell}}}$. Substituting the super block size B, the number of 1-bits $m \leq n$, and $w = O(\log n)$ into the yields the claimed asymptotic space bound. ◀

By bit-packing, we obtain the following tighter space bound.

▶ **Theorem 5.** *For a bit vector of length n, a constant-time rank and select data structure with a 2-level (or 3-level) sample tree and $\underline{\ell} \in O(1)$ levels of summary tree requires*

$$O\left(n\frac{(\log\log n)^{(\underline{\ell}-1)/2}}{(\log n)^{(\underline{\ell}-1)/2}}\right) \text{ or } \left(O\left(n\frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right)\right) \text{ bits of space}$$

on top of the space needed for the summary tree.

We prove this theorem in Section 4, where we give a detailed description of the bit-packing necessary to accomplish these space-bounds.

The Design Space. In Table 1, we summarize the results. We omit non-succinct configurations as well as configurations that would lead to higher query time (due to an increased number of levels) than other configurations that already achieve space $o(n\log\log n/\log n)$.

Among the uncompressed configurations, three seem particularly interesting as they minimize the asymptotic space-overhead for a value $\overline{\ell} + \underline{\ell}$, i.e., the total number of tree levels to be traversed. "2+3" has the smallest number of levels that achieves succinct space-overhead. "3+3" "almost" matches the space-overhead of the rank data structure, and "3+4" gets the smallest overhead among configurations with 7 overall levels. Balancing the number of levels of each type seems to be best; extreme cases do not work well. For $\overline{\ell} = 1$, no succinct configuration exists while for $\underline{\ell} < 3$, no configuration beats the overhead of the rank data structure. Among the configurations with word-packed samples, we get a similar picture. The main difference is that optimal configurations use one summary tree level less than comparable uncompressed configurations.

Table 1 Space overhead $O(n \cdot \text{table entry})$ of select data structures with $\overline{\ell}$ levels of sample tree and $\underline{\ell}$ levels of summary tree. Bold entries have the lowest space-overhead for given $\underline{\ell} + \overline{\ell}$. The uncompressed entries are from Theorem 4. The compressed entries are from Theorem 5.

	$\overline{\ell} \setminus \underline{\ell}$	2	3	4	5	≥ 7
uncompressed	2	—	$\frac{(\log\log n)^{\mathbf{1}}}{(\log n)^{\mathbf{0.5}}}$	$\frac{(\log\log n)^{1.5}}{(\log n)^{1}}$	$\frac{(\log\log n)^{2}}{(\log n)^{1.5}}$	—
	3	$\frac{(\log\log n)^{0.66}}{(\log n)^{0.33}}$	$\frac{(\log\log n)^{\mathbf{1.33}}}{(\log n)^{\mathbf{1}}}$	$\frac{(\log\log n)^{\mathbf{2}}}{(\log n)^{\mathbf{1.66}}}$	—	—
	4	$\frac{(\log\log n)^{0.75}}{(\log n)^{0.5}}$	$\frac{(\log\log n)^{1.5}}{(\log n)^{1.25}}$	—	—	—
	5	$\frac{(\log\log n)^{0.8}}{(\log n)^{0.6}}$	—	—	—	—
word-packed	2	$\frac{(\log\log n)^{\mathbf{0.5}}}{(\log n)^{\mathbf{0.5}}}$	$\frac{(\log\log n)^{1}}{(\log n)^{1}}$	$\frac{(\log\log n)^{1.5}}{(\log n)^{1.5}}$	—	—
	3	$\frac{(\log\log n)^{\mathbf{0.66}}}{(\log n)^{\mathbf{1}}}$	$\frac{(\log\log n)^{\mathbf{1.33}}}{(\log n)^{\mathbf{1.66}}}$	—	—	$\frac{(\log\log n)^{0}}{(\log n)^{1}}$ [31]
	4	—	$\frac{(\log\log n)^{0}}{(\log n)^{1}}$ [13]	—	—	—

3 Related Work

Previous work can be described using our framework as it also uses constructs similar to sample trees and summary trees. Golynski [13] uses rank information comparable to 2 levels of summary tree and 4 levels of compressed sample trees using complicated asymptotic settings for the tuning parameters. This configuration has both asymptotically higher space-overhead and needs one more level than our configuration with $\overline{\ell} = 3$ and $\underline{\ell} = 2$. Even the uncompressed version with $\overline{\ell} = 4$ and $\underline{\ell} = 2$ achieves lower space-overhead with the same number of levels. We view this as an indication that our approach of optimizing with flexible tuning parameters has considerable advantages. Raman et al. [31] use the "opposite" corner of the configuration space compared to Golynski [13] yet achieve the same lower bound. Our solution with $\overline{\ell} = 2$ needs considerably fewer summary tree levels.

While the original constant time rank data structure [17] requires just $O(n \log\log n / \log n)$ bits of space in addition to the bit vector, it utilizes two sample tree levels in addition to a lookup table. This results in poor practical performance. Providing faster and more space-efficient rank (and select) data structures has been an active line of research [12, 14, 18, 26, 33] culminating in data structures requiring just 3.5 % space-overhead [21, 34].

The first constant time select data structure [3] requires $O(n/\log\log n + \sqrt{n}\log n \log\log n)$ bits of space in addition to the bit vector. It is considered impractical due to its high constants. The work on practical select data structures shows a clear space-time trade-off ranging from very fast [33] to space-efficient [21, 34]. SIMD has been shown to be useful in practice to speed up queries in the summary tree [29]. The currently fastest select data structures by Vigna [33] use two sample tree levels. Similar designs with larger space-overhead have been long known [28]. Most of these select data structures do not provide constant time query support due to not sampling *every* position at the last level of the sample tree. For more information on practical rank-select data structures, we refer to Section 5.

We briefly mention additional results for completeness. These results concern dynamic, compressed, or information-theoretically optimal representations. First, there is a lower bound of $\Omega(\log n / \log\log n)$ amortized time for rank and select queries on dynamic bit vectors [10]. A length-n bit vector with m 1-bits requires at least $S = \lceil \log \binom{n}{m} \rceil$ bits of space. The most space-efficient compressed representation of a bit vector with constant time

rank and select queries requires $S + O(n/\text{poly}\log n)$ bits of space [30]. In practice, the RRR encoding is frequently used. Here, a bit vector with constant time rank and select support can be encoded in $S + O(n \log\log n / \log n)$ bits of space [31]. Some select data structures, e.g., [3], use an additional optimization: if a range at any level is sufficiently large, the super block of *every* 1-bit in that range is stored immediately. Furthermore, the bit vector and rank/select data structure can also be interleaved to enhance cache-locality [23].

4 Engineering Lower Bound Matching Select Data Structures

We now describe the bit-packing necessary to achieve the space-bounds given in Theorem 5. Here, we focus on the concrete representation of the data, as we describe the select query algorithm in Section 2.2. To this end, we first consider the simpler 2-level variant (Section 4.1) before extending it to the more complex 3-level sample tree (Section 4.2). Our initial descriptions give an abstract view before discussing implementation details (Section 4.4).

4.1 Word-Packed 2-Level Sample Tree

On level 2, we sample every a_2-th 1-bit. For the j-th sample, we store a tuple $\sigma_2(j) = (o, \ell)$ consisting of the offset of the super block $\sigma_2(j).o$ and the count of previously occurring large ranges $\sigma_2(j).\ell$ with the same bit-width, i.e., whose offsets require the same number of bits to represent. The *offset* is the position of the super block. For example, suppose the j-th sample has a large range $r = \sigma_2(j+1).o - \sigma_2(j).o \geq \alpha$. Then, we store the count of previously occurring large ranges with sizes $\tilde{r}$ with $\lceil \log \tilde{r} \rceil = \lceil \log r \rceil$. We call $\lceil \log r \rceil$ the *bit-width* of the range. On level 1, we sample every 1-bit up to the $(j+1)$-th sample. For each of those samples, we store the position of its super block as offset relative to the offset $\sigma_2(j).o$.

Storing the Data. These counters ℓ help us to address the samples at level 1 in the word-packed representation. For each possible bit-width of a large range, i.e., for all $w = \lceil \log \alpha \rceil .. \lceil \log n \rceil$, we have one distinct array $\mathcal{B}_w$ in which we store all samples with a corresponding bit-width consecutively ($\mathcal{B}_w$ for bottom level). Then, when answering a query, we can access the corresponding sample: Let the j-th sample have a large range with bit-width w and $\sigma_2(j).\ell = k$. Then, all corresponding samples on level 1 are in $\mathcal{B}_w[k \cdot a_2 .. (k+1) \cdot a_2)$. We store the entries of the level 2 in $\mathcal{T}$ (for top level), see Figure 3.

▶ **Theorem 6.** *For a bit vector of length n, a constant-time rank and select data structure with a 2-level sample tree and $\underline{\ell} \in O(1)$ levels of summary tree requires*

$$O\left(n \frac{(\log\log n)^{(\underline{\ell}-1)/2}}{(\log n)^{(\underline{\ell}-1)/2}}\right) \text{ bits of space.}$$

Proof. When considering an $\underline{\ell}$-level summary tree, a super block has size $B = \Theta\left(\frac{(\log n)^{\underline{\ell}}}{(\log\log n)^{(\underline{\ell}-1)}}\right)$. Additionally, there can be at most $L = \frac{n}{\alpha B}$ large ranges. Thus, on root level 2, we need $O(\log \frac{n}{B})$ bits for the offset and $O(\log L)$ bits for the count of preceding large ranges. On the lower level 1, we store the offsets of the super blocks of all a_2 consecutively sampled 1-bits per large range. For a large range of size $r \geq \alpha$, this requires $O(a_2 \log r)$ bits of space.

Combining both levels yields an overall space of $S = \frac{m}{a_2}\left(\log \frac{n}{B} + \log \frac{n}{\alpha B}\right) + \sum_{i=1}^{L} a_2 \log r_i$ bits, where m is the number of 1-bits. Since the sum of the sizes of all large ranges is at most n and due to the concavity of the logarithm, we get $S \leq 2\frac{m}{a_2} \log \frac{n}{B} + \frac{n}{\alpha B} a_2 \log \alpha$. Minimizing over a_2 by setting the derivative to zero yields $a_2 = \sqrt{2\frac{m}{n}\frac{\alpha}{\log \alpha} B \log(\frac{n}{B})}$, resulting in space $O(n\sqrt{\frac{\log n}{B}})$. Substituting $B = \Theta(\frac{(\log n)^{\underline{\ell}}}{(\log\log n)^{(\underline{\ell}-1)}})$ yields the claimed bound. ◀

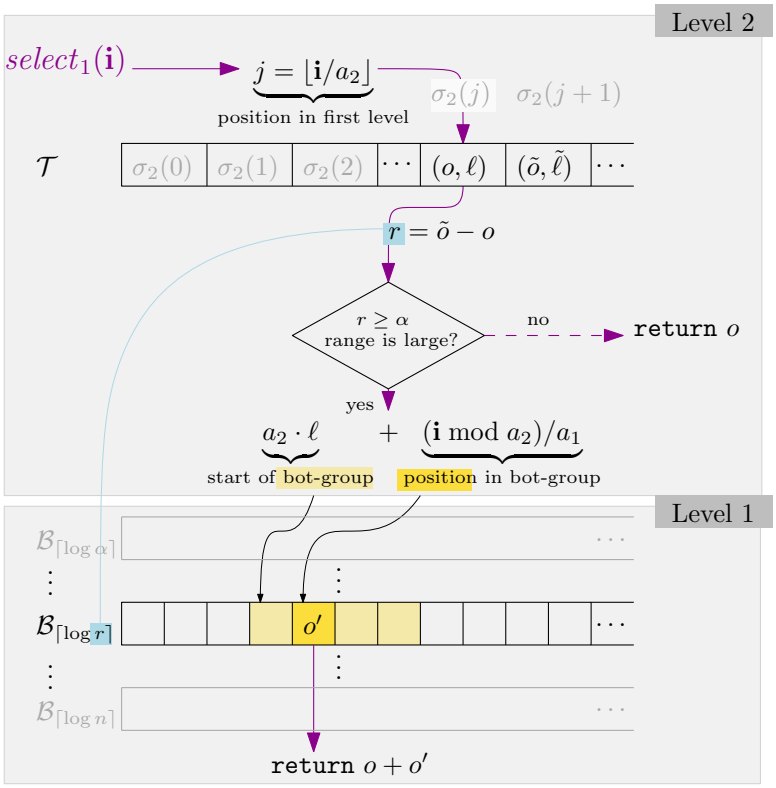

Figure 3 Path of a $select_1(\mathbf{i})$-query in a word-packed 2-level sample tree.

4.2 Word-Packed 3-Level Sample Tree

For the word-packed 3-level sample tree, we use the same techniques, i.e., counting bit-widths of ranges and storing offsets, but there is a major complication. When analyzing the space of the 2-level variant, we assume all counters $\sigma_2(\cdot).\ell$ reach their maximal value $L = n/(\alpha \cdot b_{\underline{\ell}})$. For the 3-level case, this is only feasible on level 3. At level 2, there might be too many samples to consider *all* previous ranges of the same bit-width for each sample. Thus, we have to reconsider how we store these counters at level 2.

Level 3 (Top). On level 3, we sample every a_3-th 1-bit. For the j-th sample, we store a triple $\sigma_3(j) = (o, \ell, \epsilon)$. As in the 2-level case, o is the offset of the super block and ℓ is the count of previous large ranges with the same bit-width. The additional entry ϵ is the space (in bits) required to store local counters in the next level, i.e., the bit-width of the samples at level 2 for a large range of size r is $w = \lceil \log r \rceil + \epsilon$. We store the first-level entries in $\mathcal{T}$.

Level 2 (Mid). For every large range on level 3, we sample every a_2-th 1-bit on level 2. We call these samples within the same large range on level 3 a *mid-group*. For the j-th sample, we store a tuple $\sigma_2(j) = (o', c)$. Here, o' is the offset of the super block containing the sample relative to the offset stored in the corresponding block on level 3 and c is a *local* counter. The counter is local, as it counts only the large ranges at level 2 *within* the same mid-group. For samples with bit-width w, these tuples are stored in the family of arrays $\mathcal{M}_w$, where $w = \lceil \log r \rceil + \epsilon$ as defined above ($\mathcal{M}$ for middle level).

Since the *global* count is required to answer queries correctly, we also store global counters, but just one for each mid-group. Let r' be the size of a large range at level 2 and $w' = \lceil \log r' \rceil$ it's bit-width. Next, let $\hat{w}'$ be the maximum bit-width within a mid-group. Then, we can restrict the feasible bit-widths of the samples on level 1 to $\lceil \log \alpha \rceil .. \hat{w}'$. We store these counters in the family of arrays $\mathcal{C}_{\hat{w}'}$ (for counter) that is indexed by the maximum bit-width within the mid-group. Finally, we need fast access to the global counters. To this end, we require, for each mid-group, its maximum bit-width $\hat{w}'$ and its index h among all preceding mid-groups with the same maximum bit-width. These two values are stored in the family of arrays $\mathcal{P}_w$ indexed by the bit-width of the mid-group (for pointer to the global counter).

Level 1 (Bot). For every large range on level 2, we sample every 1-bit ($a_1 = 1$). We call these samples within the same large range on level 2 a *bot-group*. For the j-th sample, we store only the offset $\sigma_1(j) = o''$ of the super block containing the sample relative to the offset stored in the corresponding block on level 2. Let w' be the bit-width of the large range on level 2 corresponding to the bot-group. Then, we store the sample in $\mathcal{B}_{w'}$ (for bottom level).

Answering a Query. We now show the path of a $select_1(i)$-query through the word-packed 3-level sample tree, see Figure 4. Starting at level 3, we access the information of the $j = \lfloor i/a_3 \rfloor$-th sample $\mathcal{T}[j] = (o, \ell, \epsilon)$ which contains the queried bit. Assuming the range of the sample is $r \geq \alpha$. Then, the bit-width of the sample on level 2 is $w = \lceil \log r \rceil + \epsilon$. On level 2, we find the sample containing the i-th 1-bit in $\mathcal{M}_w[\underbrace{\ell \frac{a_3}{a_2}}_{\text{start of mid-group}} + \underbrace{(i \bmod a_3)/a_2}_{\text{sample in mid-group}}] = (o', c)$.

Again, we assume that the sample has a large range of size r'. Thus the samples on level 1 have bit-width $w' = \lceil \log r' \rceil$. It remains to identify how many preceding bot-groups with the same bit-width w' exist. We already know the value of the local counter c. To access the global counter, we determine the maximum bit-width $\hat{w}'$ in the mid-group and the count h of preceding mid-groups with the same maximum bit-width. Both values are stored in $\mathcal{P}_w[\ell]$. Finally, we get the global counter C for the bit-width w' from the array $C_{\hat{w}'}$:

$$C = \mathcal{C}_{\hat{w}'}[h \underbrace{(\hat{w}' - \lceil \log \alpha \rceil)}_{\text{counter per maximum bit-width}} + \underbrace{w' - \lceil \log \alpha \rceil}_{\text{index of bit-width within range}}].$$

Then, we can access the offset on level 1 in $\mathcal{B}_{w'}[a_2(c + C) + i \bmod a_2] = o''$. The i-th bit is in the $(o + o' + o'')$-th super block. The concrete position within the super block is then determined using the summary tree. If at any point we do not find a large range, we return the sum of all previously found offsets.

4.3 Space Analysis (3-Level)

We first show the required space for arbitrary sampling rates a_3 and a_2 (Lemma 7). Then, we optimize these parameters regarding space consumption (Lemma 8). Finally, in Theorem 9, we show that with these sampling rates, we achieve the space bound we gave in Theorem 5. To reduce visual clutter, we use $B = b_{\underline{\ell}}$ as the size of a super block. Detailed proofs can be found in Appendix A.

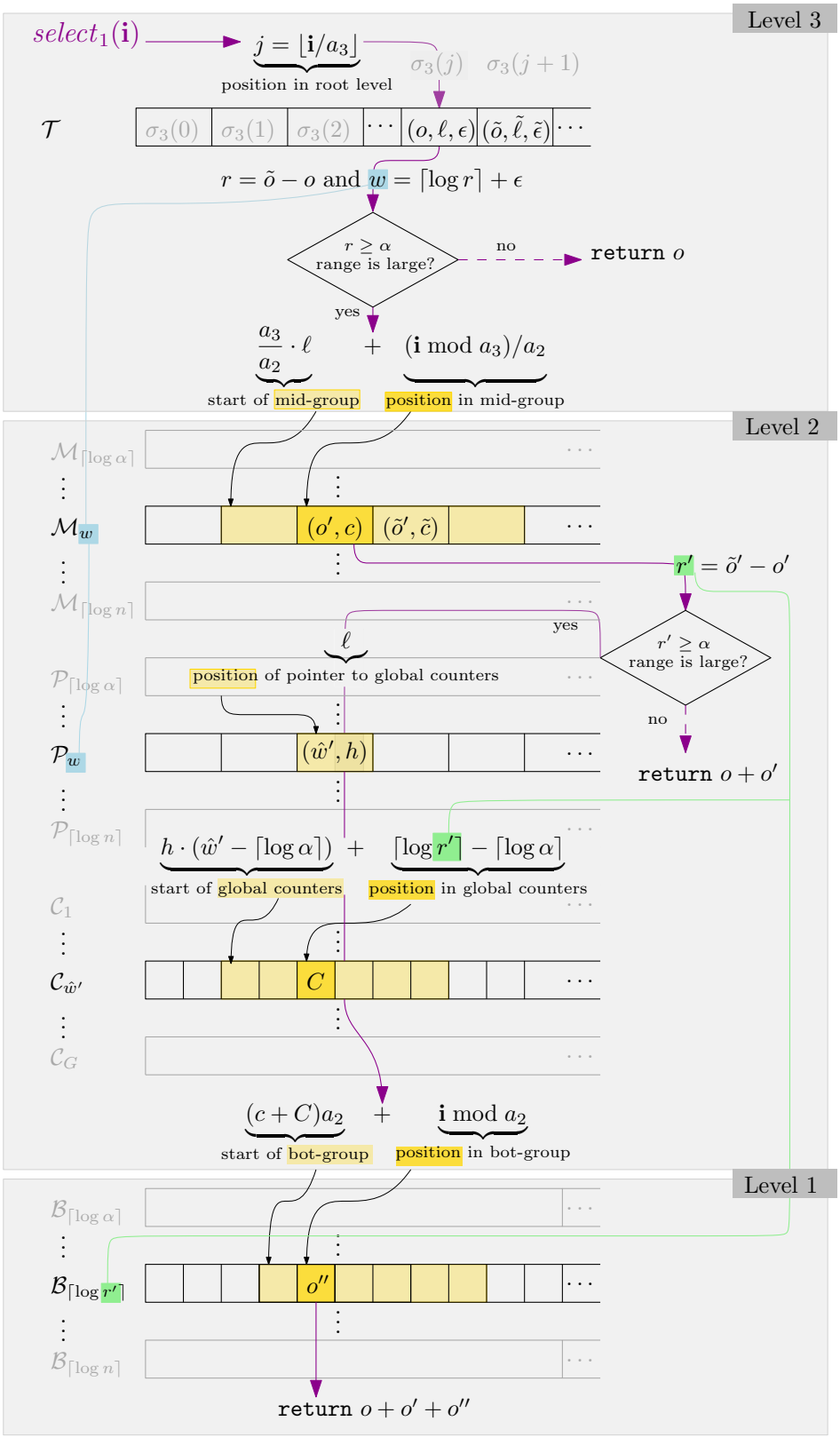

Figure 4 Path of a $select_1(\mathbf{i})$-query in a word-packed 3-level sample tree.

▶ **Lemma 7.** *Up to lower order terms, a word-packed 3-level sample tree requires space*

$$S \leq \underbrace{t\left(\log\frac{n}{B} + \log G + \log\log\frac{a_3}{a_2}\right)}_{\text{level 3}} + \underbrace{G\left(\left(\log\frac{L}{G}+1\right)\log L + 2\frac{a_3}{a_2}\log\frac{\alpha L}{G} + \log\log\frac{\alpha L}{G} + \log G\right)}_{\text{level 2}} + \underbrace{La_2\log\alpha}_{\text{level 1}},$$

where $L = \frac{n}{\alpha B}$ is the maximum number of large ranges (Observation 3), $t = \frac{m}{a_3}$ is the number of samples in level 3, and $G = \min\{L, t\}$ the maximum number of mid-groups.

▶ **Lemma 8.** *To minimize the space of a word-packed 3-level sample tree (Lemma 7) when using a $\underline{\ell}$-level summary tree, the optimal choices for a_3 and a_2 are:*

$$a_3^\star = O\left(B\frac{(\log\log n)^{(\underline{\ell}-1)/3}}{(\log n)^{(\underline{\ell}-2)/3}}\right) \text{ and } a_2^\star = O\left(\sqrt{B}\frac{(\log\log n)^{(\underline{\ell}-1)/6}}{(\log)^{(\underline{\ell}-2)/6}}\right).$$

▶ **Theorem 9.** *A word-packed 3-level sample tree on top of a $\underline{\ell}$-level summary tree requires $O\left(n\frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right)$ bits of space and can be constructed in the same time needed for the construction of the summary tree.*

4.4 Implementation Details

We implemented the word-packed $\overline{\ell}$-level sample trees as well as $\underline{\ell}$-level summary trees for $\overline{\ell} = 2..3$ and $\underline{\ell} = 2..4$. This gives us empirical results for the asymptotically lowest space overheads for given $\underline{\ell} + \overline{\ell}$, i.e., the starting points for a more structured exploration of the design space for select data structures indicated by our framework (Table 1).

4.4.1 Engineering Word-Packed Sample Trees

Our implementation of the sample trees is very close to their theoretical descriptions in Sections 4.1 and 4.2, leaving only a few details to describe here. On the root level (level 3 or 2), we store all samples in a 64-bit word each. This limits us to bit vectors of length $2^{64-\hat{\epsilon}}$, where $\hat{\epsilon}$ is the maximum number of bits necessary to store the local counters on level 2. This is a reasonable limitation, as we have shown that $\hat{\epsilon} = \lceil \log\log\frac{a_3}{a_2} \rceil$ bits suffice (Lemma 7).

To store the entries with different bit-widths in the lower levels, we use arrays where each element has bit-width w for $w = 1..64$. The underlying data is stored in 64-bit words. To access one w-bit element, we have to read at most two 64-bit words. When possible, we optimize this further for our use case by immediately returning two consecutive w-bit entries, which we require to determine whether we have a small or large range. We implemented different strategies for the computation of the sample rates a_3 and a_2, which optimize the required space, query time, or provide a trade-off. See Section 5 for empirical results. To enable efficient division, we round the sample rates to the nearest power of two.

4.4.2 Select-Optimized Summary Trees

In the theoretical analysis, we always assume to have blocks of size $b_1 = \Theta(\log n)$ on level 1 of the summary tree, as this allows us to answer rank and select queries within such a block in constant time. On modern hardware, the operations necessary to do this efficiently are available for 64-bit words. Our implementation supports $b_1 \in \{512, 1024, 2048\}$. Smaller

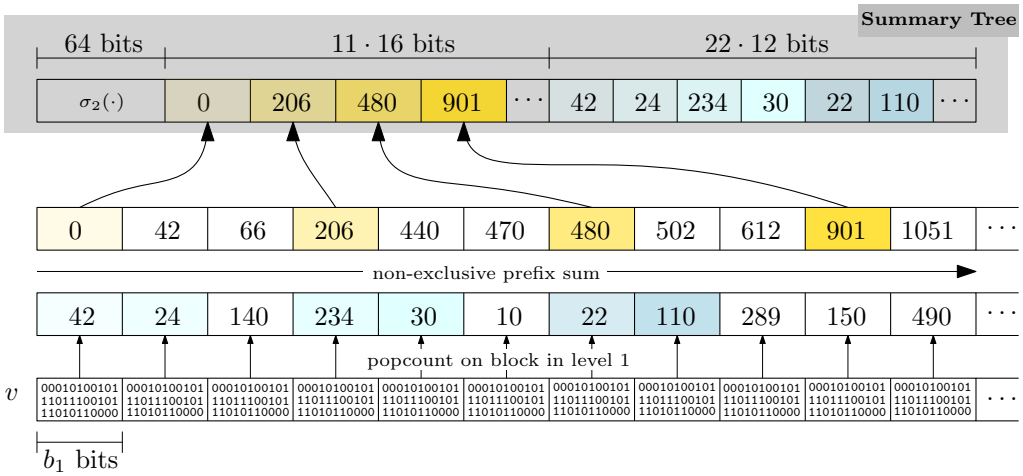

Figure 5 Structure of a summary tree as used by our implementation. We show the placement of absolute and relative counts in shades of yellow and blue, resp.

values for b_1 increase the size of the summary tree but do not increase the performance of the rank/select query on the block measurably. These values are also common in other implementations, e.g., [21, 34].

We now explain the 2-level case. For summary trees with more levels, we group and sum counters as described in Section 2.1. By using 32-bit integers for the counters, we can find the correct child block using the `_mm256_cmpgt_epi32` SIMD instruction. On level 2, we have $b_2 = 32 \cdot b_1$. For each super block j, we use 64 bits to store $\delta_2(j)$, i.e., the count of `1`-bits from the beginning of the bit vector up to that super block. On level 1, we now have to store counts for 32 blocks. For every 3rd block, we store the absolute count. For every other block, we only store the count of `1`-bits within the block. This requires 12 bits for the relative counts and 16 bits for the absolute counts, as $b_2 \leq 2^{16}$ for $b_1 = 2048$ and the final absolute count is for the 30-th block (followed by two relative counts), i.e., the maximum absolute count is less than 2^{16}. All this requires $508 = 11 \cdot 16 + 22 \cdot 12 + 64$ bits of space. For memory access we use 512 bits, which is also the size of a cache line on most x86 systems.

In Figure 5, we show the structure of the summary tree. The first 64 bits store the level 2 data. Next, we store eleven 16-bit values followed by twenty-two 12-bit values for the (absolute and relative) counts of level 1. During a select query, we can determine the closest absolute count using the SIMD instruction `_mm256_cmpgt_epi16`. Then, we only have to look at the corresponding two 12-bit values. To extract those, we only have to shift and mask three bytes. Additionally, we use 16 bits to explicitly store the number of `1`-bits preceding the first block within a super block, even though this value is always 0. While this wastes space, it removes one branch during querying.

5 Experimental Evaluation

Our experiments ran on an Intel Core i7-11700 with 8 cores with 80 KB L1 and 512 KB L2 cache (per core), and 16 MB L3 cache and 64 GB RAM. The system runs Rocky Linux v9.5. Code is compiled using rustc 1.94.0. Results on an AMD EPYC CPU are given in Appendix B.

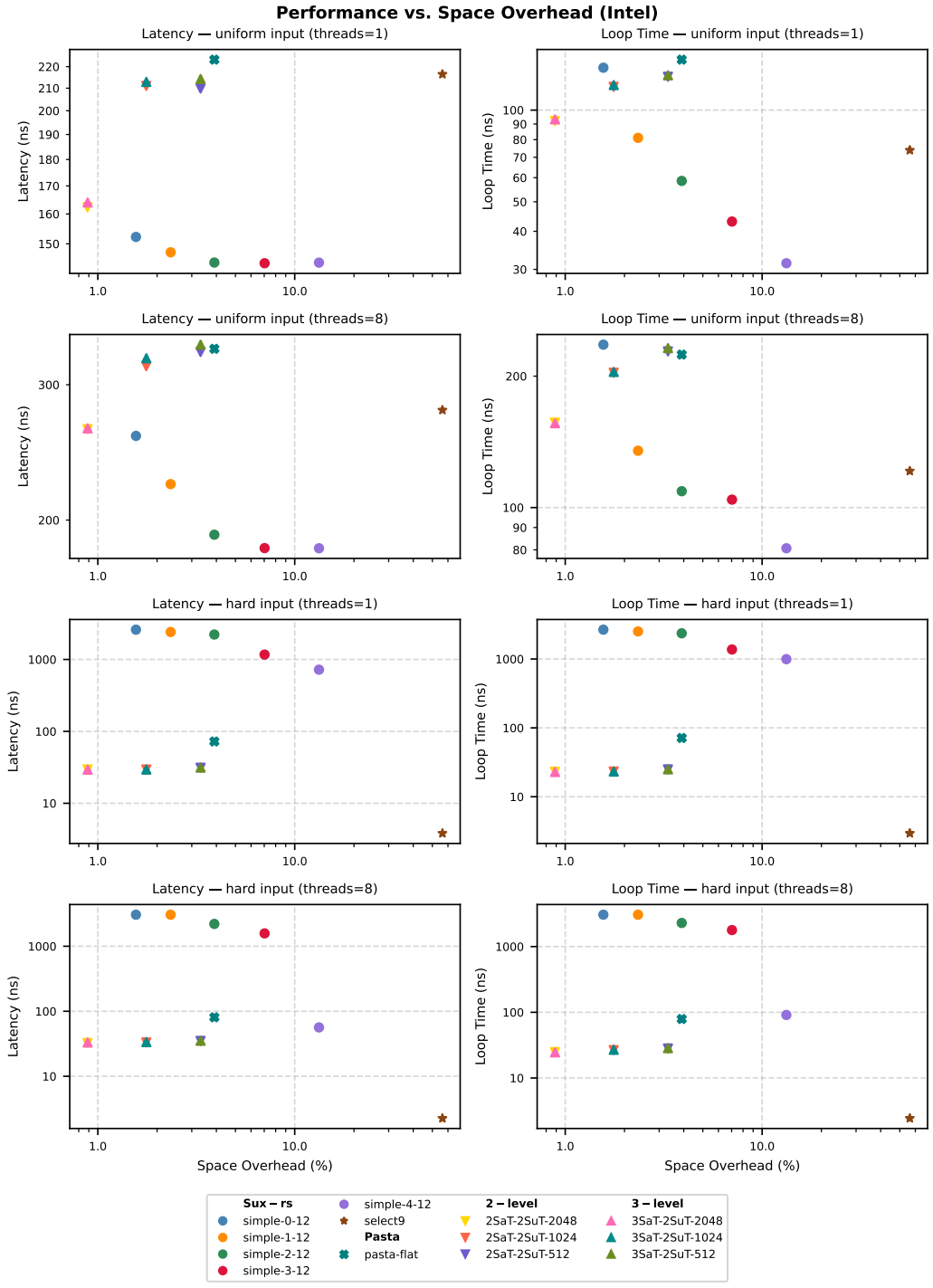

Figure 6 Time-space trade-off (log-log scale) for fill-degree 50 % on bit vectors of length 10^{10} bits.

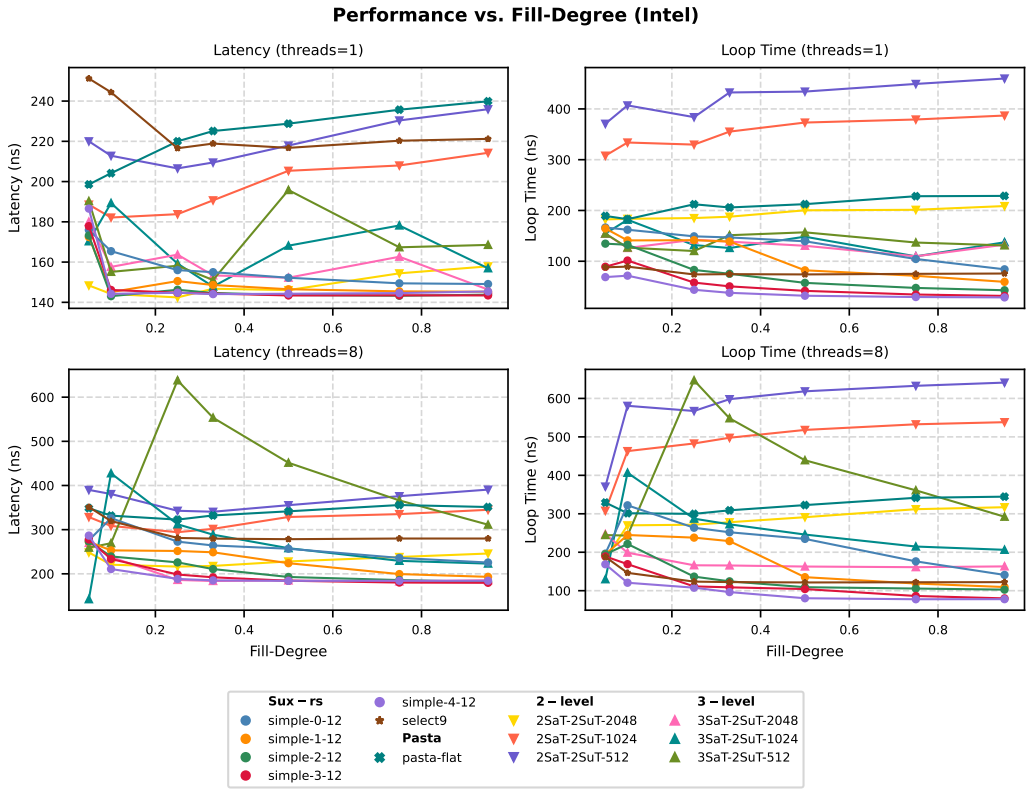

Figure 7 Query performance for different fill-degrees on a uniform bit vector of length 10^{10} bits.

Experimental Setup. We use two types of synthetic random inputs called *uniform* and *hard*. For the uniform inputs, we use a uniform distribution to set each bit to `1` with a user-specified probability. This probability is the *fill-rate* of the bit vector. The bits to query are also chosen uniformly at random. For the hard inputs, we take a uniform input and determine 10 equidistant intervals of size $5 \cdot 10^5$. All bits within these intervals are set to `0`. To make these instances hard, the queries are chosen uniformly at random but limited to small parts of the bit vector right behind an interval. This simulates queries in sparse regions, even though the bit vector is not sparse. Our fill-rates are in the range 5 %..95 %. Sparser bit vectors are better suited to compression techniques, which are outside the scope of this paper, e.g., [2]. Our measurements are the median of three runs. Each reported time is the average time of 10M queries within such a run. We ran our experiments in two settings. In the *sequential* case, only one CPU core is active at all times. In the *parallel* case, all 96 CPU cores are running the same experiment in parallel simulating memory congestion. We measure the time until all cores have finished. For each setting, we compute the latency, where queries depend on another and the loop time, where all queries are executed without dependency, allowing the CPU to pipeline instructions.

Implementations. We evaluated our implementations of the word-packed 2- and 3-level sample tree with a 2-level summary tree using blocks of size 512, 1024, and 2048 at level 1 of the summary tree. Since this paper focuses on select queries, we compare ourselves with the current state-of-the-art select data structures that are not clearly dominated by other implementations [22]. Highly engineered recent rank-only data structures [15] will generally

be faster than our rank and select data structure. We therefore do not evaluate the rank performance. In our evaluation, we include:

- **simple-0/1/2/3/4** [33] state-of-the-art $select_1(\cdot)$-only data structures from `sux-rs`,
- **select9** [33] fast rank and $select_1(\cdot)$ data structure, and
- **pasta-flat** [21] space-efficient rank and select data structure reimplemented in Rust.

Note that $select_1(\cdot)$-only data structures do not support $select_0(\cdot)$-queries and have no support for rank at all. This reduces the memory requirements by limiting the supported queries.

Experimental Results

In the space-time plots (Figure 6), we can see that our data structures can achieve very low space overhead with a quite limited performance penalty. Even with below 1 % space overhead, we are within the same ballpark performance. This is a much smaller space overhead than all the other implementations. Smaller blocks at level 1 of the summary tree are detrimental to the performance, as they increase the memory overhead resulting in more contention on the cache, see results for a CPU with larger cache sizes (Appendix B).

Our new data structures dominate *select9* which is the only competitor with worst-case constant time select. In particular, we vastly reduce space overhead while having much lower latency for easy instances. The loop times are similar. Compared to *pasta-flat*, we have similar latencies and slightly worse loop times for easy instances while being faster for hard instances. The *simple* data structures have similar latencies than ours for easy instances and better loop times. But, they are slow for hard instances and do not support rank queries.

The query performance is not strongly influenced by the fill-degree. In Figure 7, we see that in the sequential case, the latency remains rather stable. The independent queries get slightly faster for some configurations of *simple*. When running the experiments in parallel, this effect is more prominent.

6 Conclusion and Future Work

We present a new framework to classify rank and select data structures, mapping a previously unexplored design space and presenting concrete implementations in a promising region. Our data structures outperform the current state-of-the-art on hard inputs and provide a good space-time trade-off otherwise. Most notably, our data structures are more space-efficient while supporting rank *and* select queries. Future work includes further compressing both the rank and select data structure and the underlying bit vector.

References

1 Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011. `doi:10.1016/j.tcs.2010.05.003`.

2 Eric Chiu and Dominik Kempa. Engineering select support for hybrid bitvectors. *CoRR*, abs/2509.06900, 2025. `doi:10.48550/arXiv.2509.06900`.

3 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996. URL: `http://dl.acm.org/citation.cfm?id=313852.314087`.

4 Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Practical wavelet tree construction. *ACM Journal of Experimental Algorithmics (JEA)*, 26:1.8:1–1.8:67, 2021. `doi:10.1145/3457197`.

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

6 Robert Mario Fano. On the number of bits required to implement an associative memory, 1971.

7 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.

8 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

9 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

10 Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Symposium on Theory of Computing (STOC)*, pages 345–354. ACM, 1989. doi:10.1145/73007.73040.

11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *Journal of the ACM (JACM)*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Symposium on Experimental Algorithms (SEA)*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.

13 Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007. doi:10.1016/J.TCS.2007.07.041.

14 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Workshop on Experimental Algorithms (WEA)*, pages 27–38, 2005.

15 Ragnar Groot Koerkamp. Quadrank: Engineering a high throughput rank. *CoRR*, abs/2602.04103, 2026. doi:10.48550/arXiv.2602.04103.

16 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

17 Guy Jacobson. Space-efficient static trees and graphs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

18 Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In *Workshop on Experimental Algorithms (WEA)*, volume 3503 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2005. doi:10.1007/11427186_28.

19 Florian Kurpicz. Bit vector with rank and select support (part of Limnopilos). Text (visited on 2026-08-14). URL: https://codeberg.org/limnopilos/limnopilos/src/branch/main/crates/bit_vector, doi:10.4230/artifacts.27676.

20 Florian Kurpicz. Limnopilos. Software (visited on 2026-08-14). URL: https://codeberg.org/limnopilos/limnopilos, doi:10.4230/artifacts.27675.

21 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *Symposium on String Processing and Information Retrieval (SPIRE)*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. doi:10.1007/978-3-031-20643-6_19.

22 Florian Kurpicz, Niccolò Rigi-Luperti, and Peter Sanders. Theory meets practice for bit vectors supporting rank and select. *CoRR*, abs/2509.17819, 2025. doi:10.48550/arXiv.2509.17819.

23 Matthew D. Laws, Jocelyn Bliven, Kit Conklin, Elyes Laalai, Samuel McCauley, and Zach S. Sturdevant. SPIDER: improved succinct rank and select performance. In *Symposium on Experimental Algorithms (SEA)*, LIPIcs, pages 21:1–21:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SEA.2024.21.

24 Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.

25 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

26 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012. doi:10.1007/978-3-642-30850-5_26.

27 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

28 Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007. doi:10.1137/1.9781611972870.6.

29 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Inf. Syst.*, 99:101756, 2021. doi:10.1016/j.is.2021.101756.

30 Mihai Pătraşcu. Succincter. In *Symposium on Foundations of Computer Science (FOCS)*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

31 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

32 J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963. doi:10.1145/321160.321170.

33 Sebastiano Vigna. Broadword implementation of rank/select queries. In *Workshop on Experimental Algorithms (WEA)*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

34 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.

A Proof for Space Analysis of 3-Level Sample Trees (Section 4.3)

Proof of Lemma 8. We consider the parts of the space S that depend on a_3 and a_2. We have $S(a_3) = t \log \frac{n}{B} + t \log L + 2La_2 \log \alpha + t \log L + L \log L = 3 \log \frac{n}{a_3 B} + \sqrt{8} \frac{\log \alpha}{\alpha} \frac{n}{mB} \sqrt{a_3}$. We minimize it by solving $S'(a_3) \stackrel{!}{=} 0$, which results in $a_3^\star = (\frac{2m}{\sqrt{3n}} \frac{\alpha}{\log \alpha} B \log \frac{n}{B})^{2/3} = O((B \log \frac{n}{B})^{2/3}) = O(B \frac{(\log \log n)^{(\ell-1)/3}}{(\log n)^{(\ell-2)/3}})$. We do the same for a_2, i.e., $S(a_2) = (2\frac{a_3}{a_2} + a_2) \log \alpha$. We solve $S'(a_2) \stackrel{!}{=} 0$ which results in $a_2^\star = \sqrt{2a_3}$. ◀

Proof of Lemma 7. To simplify calculations, we assume that storing elements in a range $0..U-1$ requires $\log U$ bits, ignoring rounding up to $\lceil \log U \rceil$ as required for word-compression. This is admissible as a more detailed calculation only adds lower order terms to the space consumption.

We now start with *level 3*. For each of the t samples, we store the offset of the super block in $\log \frac{n}{B}$ bits. The counter c can only count up to G and thus requires $\log G$ bits of space. Last, local counters at level 2 can count at most up to $\frac{a_3}{a_2}$, thus requiring at most $\epsilon \leq \log \frac{a_3}{a_2}$ bits of space. We store this information using $\log \log \frac{a_3}{a_2}$ bits.

For *level 2*, the mid-group require $\sum_{g=1}^{G} \frac{a_3}{a_2} w_g$ bits of space in total, where w_g is the bit-width of the samples within group g. Let r be the size of the large range on the first level corresponding to a mid-group g and $\hat{c}$ the maximum local counter value in that group, then $w_g = \log r + \log \hat{c}$. Due to the concavity of the logarithm, the first term is maximized if there are many "small" large gaps, i.e., $r_g \approx \alpha$ for all mid-groups g. However, there can

be at most $G = \min\{L, t\}$ mid-groups. We have $G \leq t$, because each mid-group originates from a sample in the first level, of which there are t. We also have $G \leq L$, because there can occur at most L large ranges in the top level. Fortunately, both cases can be represented elegantly by a single expression for the size of large range, namely $\alpha L/G$. Note that if $G = L$ it correctly reduces to α, whereas for $G = t$ it correctly reduces to $\alpha L/t$.

For each mid-group g, we store the maximum bit-width in the mid-group $\hat{w}'_g$ and the count h_g of preceding mid-groups with the same maximum bit-width $\hat{w}'_g$. This requires $G \log G + \sum_{g=1}^{G} \log\log \hat{w}'_g$ bits of space. The start of the counters is at most G, i.e., the number of mid-groups. We do not further consider $\hat{w}'_g$, as it enters only as lower-order term.

Finally, we consider the space for the global counters. For each mid-group, these counters require $\sum_{g=1}^{G}((\hat{w}'_g - \lceil \log \alpha \rceil + 1) \log L$ bits of space. We can maximize the sum by maximizing $\hat{w}'_g$. However, the gaps at level 2 can never be greater than the gaps at level 1. Therefore, the sum is maximized for $\hat{w}'_g = \frac{\alpha L}{G}$.

On *level 1*, we sample every 1-bit within large ranges r' at level 2. The stored offsets then require $\log r'$ bits of space each. There are at most s entries in level 2 that have a large range. Therefore, the space for level 1 is at most $a_2 \sum_{i=1}^{L} \log r'_i$, where r'_i is the size of the i-th large range at level 2. This is maximized for $r'_i = \alpha$ for all i, yielding the claimed size. ◀

Proof of Theorem 9. We bound the *space* of the sample tree by considering all addends (Lemma 7) separately and using the space-optimal choices $a_3^\star$ and $a_2^\star$:

$$S = O(\underbrace{t \log \frac{n}{B}}_{\text{Equation (1)}} + \underbrace{t \log L}_{\text{Equation (2)}} + \underbrace{t \log\log \frac{a_3^\star}{a_2^\star}}_{\text{Equation (3)}} + \underbrace{L \frac{a_3^\star}{a_2^\star}}_{\text{Equation (4)}} + \underbrace{L \log L}_{\text{Equation (5)}} + \underbrace{L a_2^\star}_{\text{Equation (6)}}).$$

$$t \log \frac{n}{B} = O\left(\frac{n}{a_3^\star} \log \frac{n}{B}\right) = O\left(n \frac{(\log \frac{n}{B})^{1/3}}{B^{2/3}}\right) = O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{1}$$

$$t \log L = O\left(t \log \frac{n}{B}\right) \overset{\text{Equation (1)}}{=} O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{2}$$

$$t \log\log \frac{a_3^\star}{a_2^\star} = O\left(t \log\log \sqrt{a_3^\star}\right) = O\left(t \log \frac{n}{B}\right) \overset{\text{Equation (1)}}{=} O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{3}$$

$$L \frac{a_3^\star}{a_2^\star} = O\left(\frac{n}{B} \sqrt{B \frac{(\log\log n)^{(\underline{\ell}-1)/3}}{(\log n)^{(\underline{\ell}-2)/3}}}\right) = O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{4}$$

$$L \log L = O\left(\frac{n}{B} \log \frac{n}{B}\right) = O\left(n \frac{(\log\log n)^{(\underline{\ell}-1)}}{(\log n)^{(\underline{\ell}-1)}}\right) = O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{5}$$

$$L a_2^\star = O\left(L \sqrt{a_3^\star}\right) \overset{\text{Equation (4)}}{=} O\left(n \frac{(\log\log n)^{(2\underline{\ell}-2)/3}}{(\log n)^{(2\underline{\ell}-1)/3}}\right) \tag{6}$$

Analogous to the uncompressed case (Theorem 4), the *construction time* of the word-packed 3-level sample tree is bounded by the maximum number of entries in one of its level. Therefore, we can compute it in time $O\left(n \frac{(\log\log n)^{(2k-2)/3}}{(\log n)^{(2k-1)/3}}\right)$. ◀

B Additional Experimental Results

We additional ran experiments on a much more potent 96-core AMD EPYC 9684X with 64 KB L1 and 1 MB L2 cache (per core), and 1152 MB L3 cache equipped with 1536 GB (24x 64GB) DDR5-4800 memory. See Figures 8 and 9 for the results. The overall statements regarding space-time trade-offs remain the same.

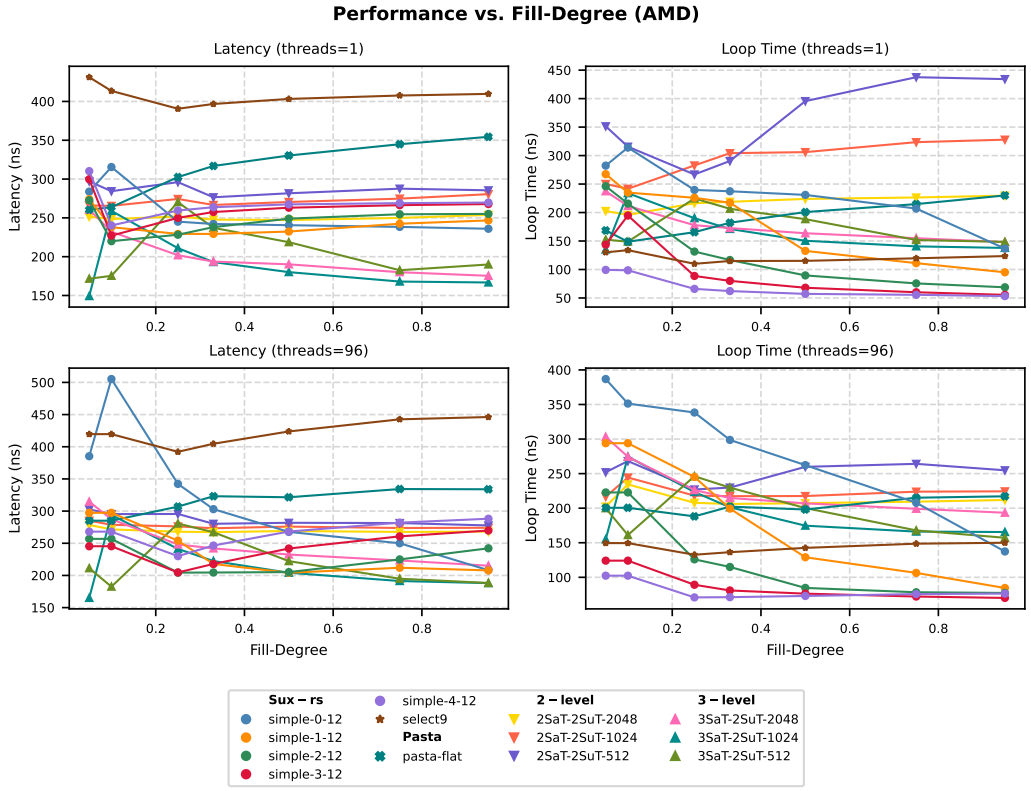

Figure 8 Query performance for different fill-degrees on a uniform bit vector of length 10^{10} bits.

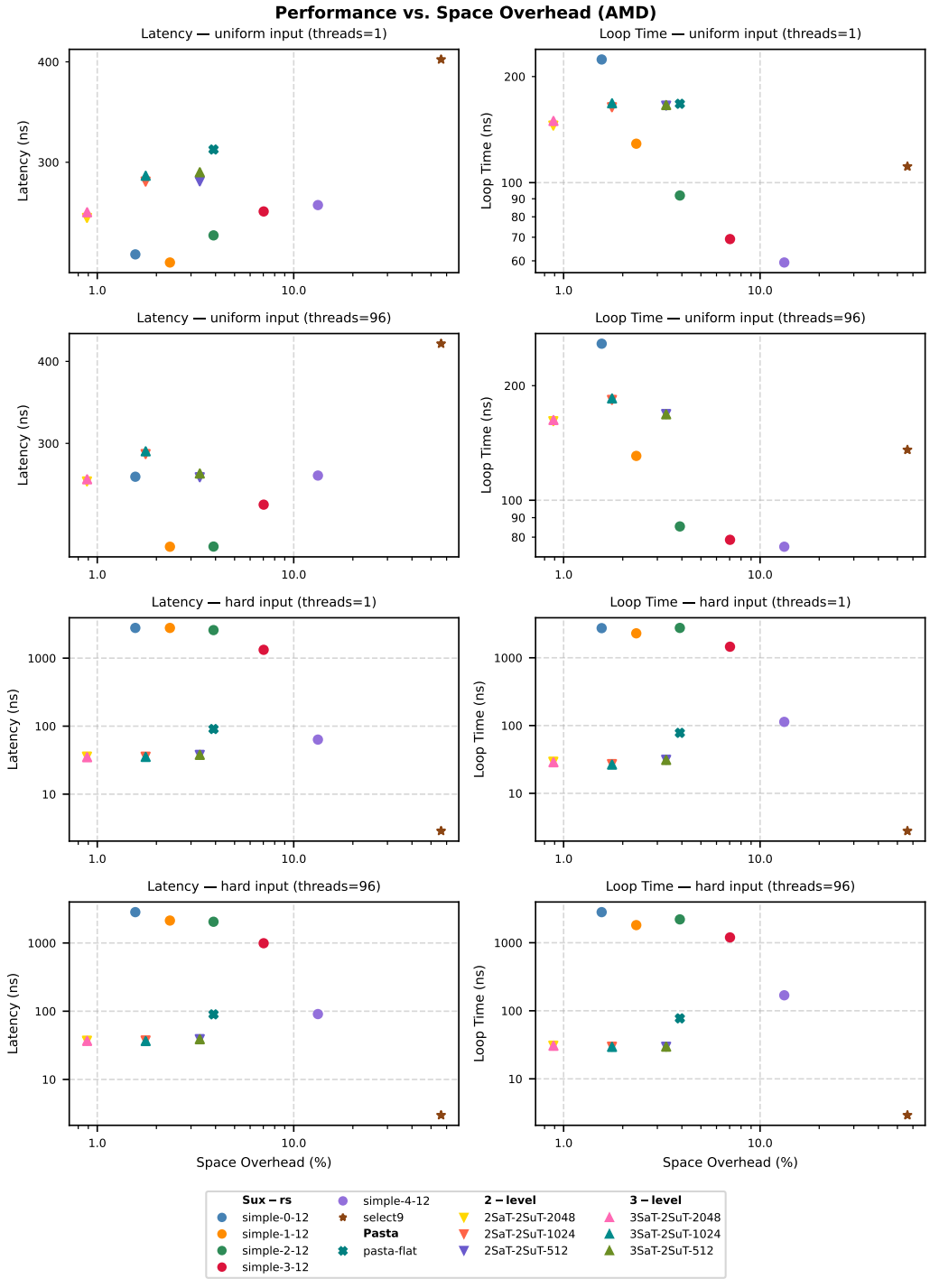

Figure 9 Time-space trade-off (log-log scale) for fill-degree 50 % on bit vectors of length 10^{10} bits.