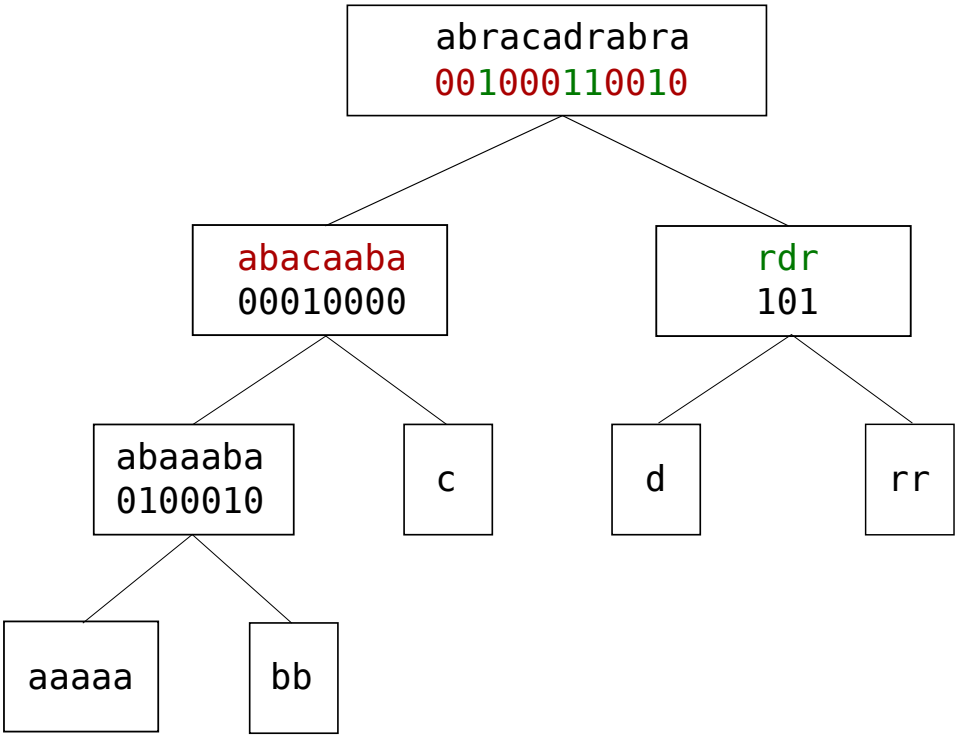
Engineering Rank and Select Queries on Wavelet Trees

Jan H. Knudsen, 20092926
Roland L. Pedersen, 20092817

Master's Thesis, Computer Science
Advisor: Gerth Stølting Brodal
June, 2015



Contents

Abstract

In this thesis we perform a survey on the applications of wavelet trees. We describe how and why modern cpu architectures give rise to certain hardware-based performance penalties. We implement a wavelet tree and measure and analyse the performance and encountered hardware-based performance penalties of building and querying the tree. Inspired by this analysis, we iteratively implement, measure, and analyse variations of the wavelet tree and its queries attempting to reduce the encountered penalties, running time and memory footprint, sometimes comparing with a theoretical analysis.

Part I
The Wavelet Tree

1 Introduction

The Wavelet Tree is a relatively new, but versatile data structure, offering solutions for many problem domains such as string processing, computational geometry, and data compression. Storing, in its basic form, a sequence of characters from an alphabet it enables higher-order entropy compression and supports various fast queries.

In this thesis we have made a short survey of some of the various applications of a wavelet tree including uses in compression and in information retrieval. We include descriptions of how the construction of a wavelet tree and its supported queries work in practice.

The practical implementation of a wavelet tree is susceptible, like all other algorithms, to the characteristics and imperfections of modern computer architectures that can degrade the performance by various penalties. We describe and analyse why these characteristics give rise to these penalties and how they impact performance.

Our focus has been to implement various variations of the wavelet tree and its queries, measuring the running times and the hardware-based penalties, and implement new variations of the wavelet tree in attempts to reduce these penalties. We also use these measurements to try and analyse and explain why the different algorithms and wavelet trees perform differently. We aim at making it something that could be useful in real world scenarios and we have tried to use inputs in our experiment that correspond to realistic use cases. We have therefore avoided impractical optimizations such as ones that require recompilation to handle different sizes of alphabets.

We have implemented and tested the construction of a wavelet tree, comparing it to the theoretical running time. We also implemented and tested the rank and select queries and performed a number of modifications, attempting to reduce the amount of hardware penalties they encounter by changing how they are calculated, changing the shape of the tree, changing what is stored and how it is stored. We test and compare these optimizations including analysing how they perform with regards to the various penalties found in modern CPUs.

We first implemented the basic construction algorithm based on the description by Navarro [1, Section 2], then expanded the implementation in various ways to attempt to improve the query algorithms.

The Wavelet Tree is a tree structure of bitmaps. It was invented by Grossi, Grupta and Vitter [2] in 2003. In its basic form, it is a balanced binary tree of bitmaps, encoding a *sequence* or *string* $S[1, n] = c_1c_2c_3 \dots c_n$ of *symbols* or *characters* $c_i \in \Sigma$, where $\Sigma = [1 \dots \sigma]$ is the *alphabet* of S, in such a way that it supports a number of fast queries on S. A balanced wavelet tree over a string S with alphabet Σ will have height $h = \lceil \log \sigma \rceil$, and $2\sigma - 1$ nodes, with σ of those as leaf nodes and $\sigma - 1$ as internal nodes. In this thesis, when we write log we actually mean $\log_2$ unless otherwise noted.

The wavelet tree supports access, rank and select queries. An access(p) query on a wavelet tree construced on string S is the query for what character c is at position p in the string S. The rank of a character c in a string S up to position p is written as $\text{rank}_p(c)$ and is defined as the number of occurrences o of c in the substring $S[0, \ldots, p]$. The position of the oth occurrence of a character c can be found with a $\text{select}_c(o)$ query.

With extensions, a wavelet tree can be used for efficient compression of S while still supporting the same queries, although not as fast. It has applications in many areas, from string processing to geometry, and can be used to represent, among others, a sequence of elements, a reordering of elements or a grid of points [1, Section 4]. When Grossi et al. [2] invented the Wavelet Tree, it was a milestone in compressed full-text indexing even though it is mentioned little in the paper. The wavelet tree has even been shown to be able to get close to a lower bound of compression called kth-order entropy encoding, and we discuss this in Section 4.2.1.

2 Related Work

The wavelet tree was first introduced in 2003 by Grossi, Gupta, and Vitter [2, Section 4.2] as a way to obtain faster rank and select query times on compressed suffix arrays while maintaining empirical entropy compression.

Gonzalo Navarro [1] explains how the wavelet tree has many and wide ranging useful applications, from string processing including compression, full-text indexes and inverted indexes to geometry processing including various queries and computations on point grids and rectangle sets as well as graphs. Gonzalo Navarro [1, Section 9] also mentions that there are other data structures that achieves better time complexity than the wavelet tree, but the wavelet tree is more practical and easy to understand and implement.

Cristos Makris [3] also describes several effective uses of a wavelet tree, including viewing it as a range searching data structure for e.g. minimum bounding volumes and effective storage compression. Using the wavelet tree as a compressing data structure is mainly about using various ways of encoding the bitmaps, such as using run-length encoding (RLE) on the bitmaps and storing the Burrows-Wheeler transformation (BWT) of the input string, or using Huffman Coding to shape the tree. The Burrows-Wheeler transformation was introduced by Burrows and Wheeler [4, Abstract] in 1994. Ferragina et al. [5, Section 2] describes in more detail how BWT can be used to reduce the problem of compressing higher-order entropy to a problem of compressing 0-order entropy, which the wavelet tree then can do using RLE. Mäkinen and Navarro [6, Section 4] invented the Huffman-shaped wavelet tree and describes in short the general principle of it without going into much detail. We describe in more detail how these methods of compression work on wavelet trees in Section 4.2.

Claude and Navarro [7, Section 2.2] give a good description of how rank and select queries is performed on the wavelet tree in practice. See Section 3.3 and Section 3.4 for our description of them, as well as Section 4.3.1 for more description of the uses of them.

Another use of a wavelet tree is answering Range Quantile queries and is described by Gagie et al. [8, Section 3]. See Section 4.3.2 for a description of this.

Julian Shun [9] describes various parallelized algorithms for constructing the wavelet tree by utilizing the GPU, achieving up to a 27x speedup over the sequential construction algorithm.

Alex Bowe [10] describes how a multiary wavelet tree can be combined with an RRR structure to support faster queries than a binary wavelet tree can accomplish using an RRR structure invented by Raman et al. [15]. An RRR structure allows computation of binary rank in $O(1)$ time and provides zero-order entropy compression for binary strings. We found this result late, otherwise we would have implemented and tested it. By using a multiary wavelet tree with the RRR structure it is possible to achieve higher order entropy compression.

3 The Wavelet Tree

3.1 Constructing the Wavelet Tree

An example of a Wavelet Tree can be seen in Figure 1.

The wavelet tree is constructed recursively, starting at the root node and moving down the tree, with each node in the tree receiving a string constructed by its parent, except the root node that receives the full input string. Let S_{parent} of length n_{parent} be the string passed to the node from the parent node or, in the case of the root node, the input string to the wavelet tree itself. Σ_{parent} is the alphabet over which S_{parent} is defined where each entry in the alphabet, a *character* or *symbol*, has a position in the alphabet. The size of Σ_{parent} is σ_{parent}. Each node stores a bitmap of size n_{parent} as well as pointers to its left and right child nodes.

Each node calculates the middle character of Σ_{parent} and uses it to set the bits in the bitmap and split S_{parent} in two substrings S_{left} and S_{right}, passing those on to the left and right child nodes.

Let $i = \left\lfloor \frac{\sigma}{2} \right\rfloor$ be the index of the middle of the alphabet Σ_{parent}. S_{left} is then the subsequence of S_{parent} formed by the characters $c \in S_{parent}$ where $c \in \Sigma_{parent}[1 \dots i] = \Sigma_{left}$. S_{right} is the subsequence of S_{parent} formed by the characters $c \in S_{parent}$ where $c \in \Sigma_{parent}[i+1 \dots \sigma_{parent}] = \Sigma_{right}$. Alternatively, S_{left} can be considered to be the subsequence of S_{parent} where all characters $c \in \Sigma_{right}$ have been stripped out. Similarly S_{right} can be considered the subsequence of S_{parent} where all characters $c \in \Sigma_{left}$ have been stripped out. This also means that the alphabets for the substrings S_{left} and S_{right} do not overlap, that is $\forall c \in \Sigma_{left} : c \notin \Sigma_{right}$ and vice versa. The characters in the subsequences S_{left} and S_{right} occur in the same order they do in S_{parent}. All these strings are not stored anywhere in the wavelet tree. They are only used for the construction of the tree, but can later be reconstructed from the information stored in the bitmaps if need be.

Each bit in the bitmap corresponds to a character in the string S_{parent}. If a character c at position p in S_{parent} is in the left side of the alphabet Σ_{parent}, that is $c \in \Sigma_{left}$, the bit in the bitmap at position p will be set to 0. If instead $c \in \Sigma_{right}$, the bit at position p will be set to 1. Assuming the alphabet is in sorted order with regards to the *greater*

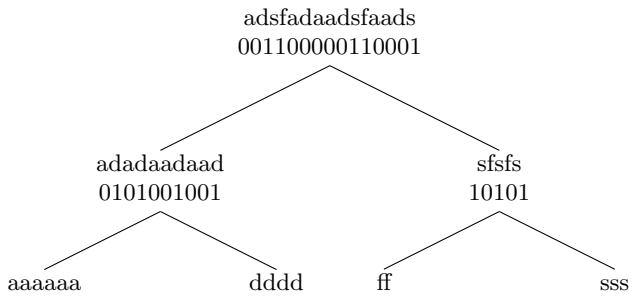


Figure 1: Wavelet Tree on string *adsfadaadsfaads* with alphabet $\Sigma = [adfs]$. Note that only the bitmaps are actually stored in the tree. The characters are annotations for ease of understanding.

than ($>$) comparison operator, this can be computed as $c \in \Sigma_{right} = c > \Sigma[\lfloor \frac{\sigma}{2} \rfloor]$. If the alphabet is not in sorted order, either lookups into the alphabet list or a mapping to and from an alphabet in sorted order will be needed to calculate whether a given character is on the left or right side of the alphabet. The leaf nodes of a wavelet tree will appear in the same order as the characters they represent appear in the alphabet used.

This process continues recursively in each child node except where only one character is left in the alphabet of the input string of a node, $\sigma_{parent} = 1$. That node is then considered a leaf node and needs not store a bitmap. Each node in a wavelet tree can be considered a full wavelet tree for the string S_{parent} it was passed from its parent node.

At each level in the tree at most n bits are stored in the bitmaps in total, making $n \cdot h = n \cdot \log \sigma$ an upper bound to the total number of bits that a wavelet tree stores in its bitmaps. In addition to this, each node takes some constant amount of machine words of space, and there are $2\sigma - 1$ nodes in the tree. ws is the size of our machine words. This makes the total memory consumption $O(n \log \sigma + \sigma \cdot ws)$ bits.

The Wavelet Tree can theoretically be constructed in $O(n \cdot h) = O(n \log \sigma)$ time as the sum of the lengths of the strings being processed at any single layer of the tree is the length of the input string to the tree.

The pseudo-code for the Wavelet Tree node construction algorithm is shown in Algorithm 1. It is recursively defined, calling itself to construct the left and right sub-tree from the root node and down. At each recursion the algorithm splits the given alphabet in two halves and traverses the given string putting each character into a left or right partition based on whether the character was in the left or right half of the alphabet.

3.2 Access Query

An access(p) query is the query for what character c is at position p in the string S the wavelet tree is constructed for. The query can be answered by a single downward traversal of the wavelet tree. Starting at the root node, an access query will look up the bit b at position p in the bitmap of the root node. If b is 0, it knows that $c \in \Sigma_{left}$ and must therefore traverse into the left child node. If b is 1, it means that $c \in \Sigma_{right}$ and the algorithm should traverse into the right child node instead. Before the algorithm

Algorithm 1 Construction of nodes in the Wavelet Tree

```
function ConstructNode(S, Σ)
    if |Σ| = 1 or |S| = 0 then
        return Self
    end if
    (Σ_left, Σ_right) ← Σ
    SplitChar ← Σ_left[σ_left]
    for all c in S do
        if c > SplitChar then
            S_right.Append(c)
            Self.Bitmap.Append(1)
        else
            S_left.Append(c)
            Self.Bitmap.Append(0)
        end if
    end for
    RightNode ← ConstructNode(S_right, Σ_right)
    LeftNode ← ConstructNode(S_left, Σ_left)
    return Self
end function
```

can continue down into the child node, it must know what position in the left (or right) substring S_{left} (or S_{right}) the character c has been mapped to.

If b is 0, the position of c in S_{left} is the number of occurrences of 0 in the bitmap up to position p. If b is 1, the position of c in S_{right} is the number of occurrences of 1 in the bitmap up to position p. This is also called $\text{rank}_0(p)$ or $\text{rank}_1(p)$ or the *binary rank* of 0 or 1 in the bitmap up to position p. In the most basic way *binary rank* can be calculated using a linear scan of the bitmap in $O(n)$ time, and since it is calculated one per level of the tree, the access query time becomes $O(n \cdot h) = O(n \log \sigma)$. The result of the binary rank is used as the position p in the child node we traverse into. The traversal continues until it reaches a leaf node which then corresponds to the character at the original position p parameter of the query. The character is then returned. Later in this thesis we will work on improving the running time of the binary rank query (see Section 8.1.1).

We have chosen not to implement or test access queries on our implementations of a wavelet tree. We have done this to reduce the amount of code and testing needed and because the behaviour of rank (see Section 3.3) and access queries are so similar because they both use binary rank. Our optimizations to binary rank can also be used for access and because of this we implement and test only the more complicated rank query.

3.3 Rank Query

The rank of a character c in a string S up to position p is written as $\text{rank}_p(c)$ and is defined as the number of occurrences o of c in the substring $S[0, \ldots, p]$.

The rank query on a wavelet tree starts from the root of the wavelet tree and moves down through the tree until it hits the leaf node corresponding to the input character, much like the access query. Also like the access query, each node calculates the binary rank of a character in the bitmap of the node and it is used as the positional parameter in the child node. Unlike the access query, the rank query is looking at a specified character c up to a position p, and whether it is the left or the right child node that is traversed into is decided by what c is represented as in the bitmap of the current node and is calculated like it is when constructing the tree.

When the leaf node is reached, the binary rank calculated in the parent node is the rank of the input symbol up to the original input position. Intuitively this makes sense because leaf nodes correspond to only one character and the rank of a character up to a position in a string containing only that character is the same as the position. Figure 2 shows an example of how this concept works. In the example, the rank query looks for the number of occurrences o of the character $c =$ *'a'* up to position $p = 10$. It begins at the root and queries recursively towards the leaf node corresponding to *'a'*. In each recursive call p is set to o_{parent} because only the 0s (or 1s) are mapped to the child node and o indicates how many of these correspond to characters occurring before the original position p_{parent}. In all the bitmaps *'a'* is represented as 0 and in the root there are 6 occurrences of 0 up to position 10. In the left child node the algorithm then counts the number of 0s (3, making $o = 3$) up to position 6 ($p = 6$), and in the leaf it counts the number of 0s until position 3 of which there are 3 ($o = 3, p = 3$). This means that there are 3 occurrences of *'a'* in S up to position 10.

We have written the rank query as pseudo-code in Algorithm 2 using an object-oriented approach.

Algorithm 2 Rank of character c until position p

```
function RANK(c, p)
    if Self.IsLeaf then
        return p
    end if
    CharBit ← bit representing c in bitmap of current node
    o ← BINARYRANK(CharBit, p)
    if CharBit = 1 then
        Rank ← RightChildNode.RANK(c, o)
    else
        Rank ← LeftChildNode.RANK(c, o)
    end if
    return Rank
end function
```

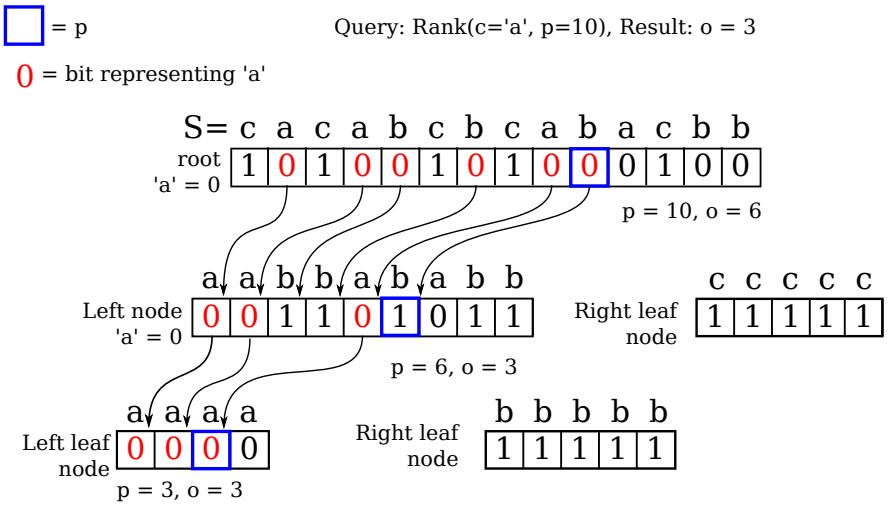


Figure 2: Rank: This figure shows how the rank query algorithm works. In this example the algorithm looks for the number of occurrences o of $c =$ *'a'* until position $p = 10$. It begins in the root and queries recursively towards the leaf node corresponding to *'a'*. The arrows indicate a mapping between the bitmaps. In each recursion $p = o_{parent}$. Each of the 0s before p_{parent} in the parent node maps to bits before p the child node. In the root there are 6 0s up to position 10. In the left node there are 3 0s up to position 6 and in the leaf there are 3 0s up to position 3. This means there are 3 *a*s up to position 10 in S.

3.4 Select query

The position of the oth occurrence of a character c can be found with a $\text{select}_c(o)$ query. The ith occurrence can be found with a traversal up through the wavelet tree starting at the leaf node corresponding to the character c and ending at the root node. This means it is necessary to find the leaf node corresponding to c.

The leaf node corresponding to a character c can be found by a downward traversal of the tree, from the root to the leaf node without accessing any of the bitmaps. Which child node, left or right, should be traversed is determined by computing whether $c \in \Sigma_{left}$ or $c \in \Sigma_{right}$. If $c \in \Sigma_{left}$, the traversal continues in the left child node, otherwise it continues in the right child node. The traversal continues until a leaf node is reached as that will be the leaf node corresponding to c.

After having found the leaf node the algorithm turns to do the upward traversal to find the position of the ith occurrence of c in the bitmap of the root node. This is done by finding the oth occurrence of 1 or 0, the bit representing c, in the bitmaps from the found leaf node to the root node, the oth occurrence of 1 or 0 being the bitmap entry corresponding to the ith occurrence of c in S. For a node v, o is the position of the bit representing c in the bitmap of the child node of v that contains c. Occurrence o is calculated for each node during the upward traversal and o increases (or at least does not decrease) as each bitmap higher up in the tree corresponds to more and more of the original input string. To know which bit, 1 or 0, to look for the oth occurrence of, the

algorithm must know which bit c has been mapped to in those bitmaps. Starting at the leaf node corresponding to c, which bit in the bitmap of the parent node that represents all characters in this node, among them c, can be computed by comparing the left or right child node pointer of the parent node with the address of this node. If the right child pointer of the parent node points to the current node, then the current node is the right child of its parent node and c and the rest of the characters in this node will be represented by a 1 in the bitmap of the parent node, otherwise it will be represented by a 0.

Having found the bit representing c in the parent node, the algorithm looks for the position of the oth occurrence of that bit in the bitmap of the parent. This is also called $\text{select}_1(o)$ or $\text{select}_0(o)$ or the *binary select* of 1 or 0 of the bitmap. It can be implemented as a linear scan of the bitmap, but this is inefficient and later in this thesis we will look at how to improve the running time of binary select. The position of this occurrence is then the new o parameter for the next step up the tree.

In Algorithm 3, we display pseudocode for select queries. GetLeaf is the function performing the initial downward traversal to find the leaf node corresponding to the character c. SelectRec is the function performing the upward traversal, finding the (varying) oth occurrences of 1 or 0 in the bitmaps up the tree, in a recursive manner. Select is the function computing the $\text{select}_c(o)$ query on the wavelet tree. It initiates the downward traversal via GetLeaf and then the upward traversal via SelectRec.

An example of how the intuition behind why Select works is shown in Figure 3. In the example the algorithm looks for the position of the 3rd occurrence of *'a'*. It looks for either 0 or 1 based on how *'a'* is represented in the bitmap of the current node. In this example *'a'* is always represented as 0. Select starts in the leaf of *'a'* where $p = 3$ and $o = 3$ and moves recursively towards the root. In each recursive call $o_{parent} = p_{child}$, meaning that p_{child} becomes the occurrence Select looks for in the parent. In this example Select therefore looks for the position of the 3rd occurrence of 0 in the parent of the leaf, which is 5. In the root it then looks for the position of the 5th occurrence of 0 which is 9 corresponding to the position of the 3rd a in S.

4 Applications

4.1 What The Wavelet Tree Can Represent

The Wavelet Tree has multiple applications that each utilize the wavelet tree differently and use it for storage of, and queries on, different types of data. These applications use the wavelet tree to achieve different representations which can be split into three main types: A sequence of values, a reordering or permutation, and a grid of points.

Using the Wavelet Tree to store a sequence of values is perhaps the most basic way to utilize the tree. The Wavelet Tree stores the sequence and supports access, rank, and select queries on the sequence.

The Wavelet Tree can also be used to describe a stable reordering of the symbols in a string S, *stable* meaning that the relative order of entries of the same symbol remain

Algorithm 3 Select

```
function Select(c, Occurrence)
    Leaf ← GetLeaf(c)
    if Leaf is a right child node then
        CharBit ← 1
    else
        CharBit ← 0
    end if
    return Leaf.Parent.SelectRec(CharBit, Occurrence)
end function

function SelectRec(CharBit, Occurrence)
    Position ← BinarySelect(CharBit, Occurrence)
    if Self is the root node then
        return Position
    end if
    if Self is a right child node then
        CharBit ← 1
    else
        CharBit ← 0
    end if
    return Parent.SelectRec(CharBit, Position)
end function

function GetLeaf(c)
    if Self.isLeaf then
        return Self
    end if
    if c ∈ Σ_right then
        return RightChild.GetLeaf(c)
    else
        return LeftChild.GetLeaf(c)
    end if
end function
```

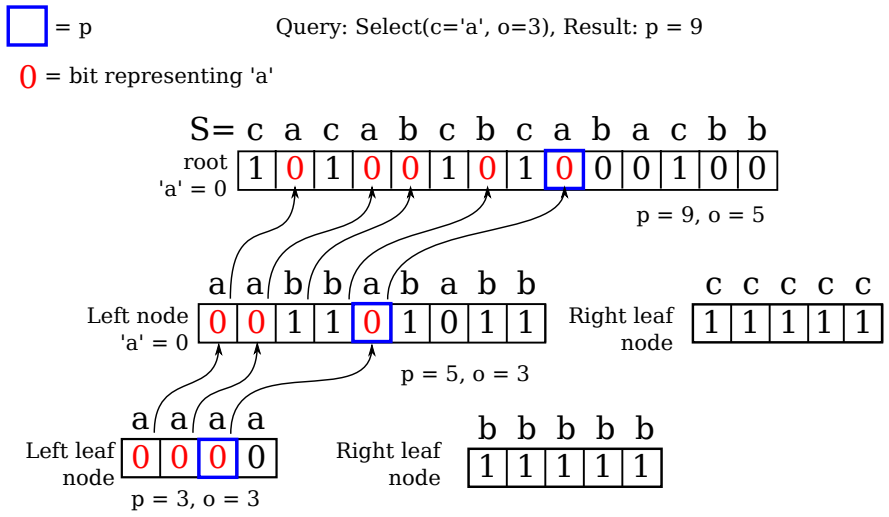


Figure 3: Select: This figure shows how the select algorithm works. In this example Select looks for the position of the 3rd occurrence of a which is represented as 0 in the leaf. Select starts in the leaf of a where $p = 3$ and $o = 3$ and moves recursively towards the root. Along the way Select looks for the position of the 3rd occurrence of 0 in the parent of the leaf, which is 5, $(p = 5, o = 3)$ and in the root it then looks for the position of the 5th occurrence of 0 which is 9, $(p = 9, o = 5)$ corresponding to the position of the 3rd a in S.

the same. This property can be relevant e.g. when using key-value pairs where the order of values matters even when the keys are identical. This also means that if the leaves are traversed, with all the occurrences of the smaller symbols found first, then all the symbols within a leaf are ordered by their position in the original string. This means that the leaves of the symbols appear in ascending sorted order from left to right in the tree. If one then has a permutation of a string e.g a string sorted in descending order and stores it in a wavelet tree, it is then possible to access the symbols in either ascending or descending order based on whether the symbol is tracked downwards through the tree until the corresponding leaf is found, or whether the symbol is tracked upwards from the leaf. The downward tracking would then result in an ascending order and the upward tracking would result in descending order. The wavelet tree is therefore able to represent a reordering of a string and the order is based on how the alphabet is sorted.

A Wavelet Tree can also represent an $n \times n$ grid of n points where no two points share the same row or column. One can map a general set of n points to such a discrete grid and then store the real points somewhere else. If we have points sorted by the x-coordinate and take only the y-coordinates $S_y[1, n] = y_1, y_2, ..., y_n$ and save S_y in a Wavelet Tree we can the find the ith point in x-coordinate order by accessing the corresponding y-coordinate in the wavelet tree. If we want the ith point in y-coordinate order we can access the leaf of a given y-coordinate and find its corresponding x-coordinate by querying up through the tree until we find the original position of y in S. The corresponding x-coordinate will be at the same position. Querying from a leaf gives the

Definition 1. *:* Entropy

Let S be a sequence of n symbols from an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$ with cardinality σ. Then entropy H is defined as

$$H = \sum_{i=1}^{\sigma} p_i \log \frac{1}{p_i} ,$$

*where p_i is the probability of the i*th *symbol in the alphabet appearing in S.*

points in y-coordinate order because the leaves are sorted by y-coordinate. The purpose of storing an $n \times n$ grid this way using a wavelet tree is to be able to find points within a rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ in order to for instance be able to do two-dimensional range search queries in $O(\log n)$ time. This running time can be improved to $O(\frac{\log n}{\log \log n})$ using $O(n \log n)$ bits and this running time cannot be improved within space $O(n \log^{O(1)} n)$ [1, Section 7.1]

4.2 Compression

The Wavelet Tree has many uses for compression of data [1]. Some of the main compression techniques are different ways of encoding the bitmaps and changing the shape of the wavelet tree [1, Section 3].

The main advantage of the wavelet tree with regards to compression is that it supports entropy bounds in the attained space complexity of the various wavelet tree compression methods [3, Section 2.1].

4.2.1 Entropy

Cristos Makris [3, Introduction] gives a definition of entropy as found in Definition 1.

Entropy represents a lower bound to the average number of bits needed to represent each symbol in S according to the coding theorem of Shannon [3, Introduction] and is the bound that compression researchers compare their results to.

This theoretical definition of entropy is often replaced in scientific literature by a more practical definition: *empirical entropy*. There are two versions: empirical zero-order entropy H_0 and empirical kth-order entropy H_k, and they are defined in Definition 2 and Definition 3. H_k takes into account a context of size k of the symbol appearances, i.e. the suffixes of length k of each symbol appearance in the string, while H_0 does not and treats symbols independently instead.

The entropy H_k often defines a lower bound for bit space usage that is smaller than the lower bound of H_0 [5, Section 2].

There is a number of ways to achieve kth-order or zero-order entropy compression in a wavelet tree, the details of which is described later. The techniques used include the

Definition 2. *:* Empirical zero-order entropy, H_0
Let S be a sequence of n symbols from an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$. The entropy H_0 is defined as

$$H_0 = H_0(S) = \sum_{c_i \in \Sigma} \frac{n_i}{n} \log(\frac{n}{n_i})$$

where n_i is the number of appearances of character c_i in S.

Definition 3. *:* Empirical *kth*-order entropy, H_k
For a string $w \in \Sigma^k$ let us define w_S as the concatenation of characters that follow w in S. Then the kth-order empirical entropy of S, is defined as follows

$$H_k = H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| H_0(w_S)$$

Burrows-Wheeler Transformation (see Section 4.2.3), using Run-Length Encoding (see Section 4.2.2), and Huffman-Shaping the wavelet tree (see Section 4.2.4)

Using the Burrows-Wheeler transformation on the input we can reduce the problem of achieving H_k compression to achieving H_0 compression. In other words, if we have a good compression algorithm that achieves compression within the H_0 lower bound, then by using that algorithm on the Burrows-Wheeler transformation of the input we can achieve compression within the H_k lower bound [5, Introduction]. The problem for a long time was that there existed no good way to achieve compression within the H_0 lower bound or at least it was a problem before the wavelet tree was invented [5, Introduction].

To achieve zero-order entropy a Huffman shaped wavelet tree can be used [6, Section 4]. Claude and Navarro [7, Section 3] describes a way to also have zero-order entropy space usage for large alphabets. It is therefore possible to get space usage within zero-order entropy even for large alphabets using the wavelet tree. Huffman shaping does not care about how symbols are grouped but only looks at their frequency of appearance. Because of this building a Huffman shaped wavelet tree on the Burrows-Wheeler transformation of a string is not different from building it using the original string. Zero-order entropy can also be achieved by run-length encoding the bitmaps in the wavelet tree, which is an approach that can be used when compressing the Burrows-Wheeler transformation of the input string using the wavelet tree [5, Introduction (**B**)]. Run-length encoding takes symbol grouping into account and this means that using a combination of run-length encoding of bitmaps and taking the Burrows-Wheeler Transformation of the input string and using the wavelet tree it is possible to achieve compression within the lower bound of kth-order entropy.

4.2.2 Run-Length encoding

Run-length encoding (RLE) is a simple process where the number of consecutive repeats of each symbol is stored instead of storing the symbols themselves. If we have the string *aaaaacccaaaaabbbaa* we can run-length encode this to *a5c3a5b3a2* which is a smaller string containing the same information. It is necessary to store the symbol and its number of consecutive repeating occurrences because we need to be able to identify which symbol occurs where and how many times in order to be able to reproduce the original string. The longer the sequence of a repeating symbol is, the less space is used since it can be stored as one number plus the related symbol.

When representing the string using a wavelet tree, the problem gets reduced to run-length encoding a string of bits (the bitmap in each node). Since a binary number only has an alphabet of size two it is not necessary to store both the symbol and its occurrence but only the occurrence, if we adopt the convention that the first number is always the amount of 0s and the second number is always the amount of 1s, continuing this trend for the entire string so that even-index numbers correspond to 0 and odd-index to 1. As an example, if we look at the bitmap of an input string *aaaaacccaaaaabbbaa* which is *000001110000000000* when stored in a wavelet tree, then it can be encoded and stored as the numbers 5 3 10. Figure 6a shows an example of a wavelet tree with run-length encoded bitmaps.

If we do not consider how a computer saves numbers but only consider the amount of length-encoding numbers that needs to be stored then our example RLE compression of 000001110000000000 achieves a great reduction in space. From 18 numbers to only 3 numbers, which contain the same information.

If we do consider how a computer saves numbers then the reduction is not that great because if we assume that each number is represented as an integer, then the run-length encoded bitmap uses more space than just storing the original bitmap. This is because an integer uses 4 bytes of space which is 32 bits and we need to store three integers giving us a total of $32bits \times 3 = 96bits$ which is significantly larger than just the 18 bits we need to store for the original bitmap assuming we can store 1 and 0 using only 1 bit[1]. This means that the symbols in the string need to, on average, repeat consecutively more than 32 times before RLE achieves better space usage than just storing the bitmaps. This is assuming 4-byte integers are used to store the RLE values. To be able to support storing the RLE of any bitmap, even one containing only 0s or only 1s, the RLE values should be able to store as high a value as the bitmap is long, which might require more bytes per value. A 32-bit unsigned integer supports storing a value up to $2^{32} = 4\,294\,967\,296$. Alternatively, variable-length encoding of the numbers can be used. The limit on bitmap length is also the maximum supported length of input string for a wavelet tree.

RLE is still useful despite of this limitation when using fixed-length encoding because you usually want to compress massive amounts of data and if that data uses an alphabet that is small enough then, as previously stated RLE can achieve compression close to the zero-order entropy when working with binary alphabets.

[1]This can be accomplished using C++ and Vector<bool>

$$\begin{bmatrix} bananahat\# \\ ananahat\#b \\ nanahat\#ba \\ anahat\#ban \\ nahat\#bana \\ ahat\#banan \\ hat\#banana \\ at\#bananah \\ t\#bananaha \\ \#bananahat \end{bmatrix} \Rightarrow \begin{bmatrix} \#bananaha\mathbf{t} \\ ahat\#banan\mathbf{n} \\ anahat\#ban\mathbf{n} \\ ananahat\#\mathbf{b} \\ at\#banana\mathbf{h} \\ bananahat\# \\ hat\#banan\mathbf{a} \\ nahat\#ban\mathbf{a} \\ nanahat\#b\mathbf{a} \\ t\#bananah\mathbf{a} \end{bmatrix}$$

Figure 4: Example of a Burrows-Wheeler transformation of the string *bananahat*

If the Burrows-Wheeler transformation is applied to the string before it is saved in the wavelet tree and run-length encoded then the number of consecutive repeats of a symbol is increased which enables even greater compression.

4.2.3 Burrows-Wheeler transformation

The Burrows-Wheeler Transformation (BWT) transforms a string S into a string of the same length with the same characters with the characteristic that characters are grouped into runs of similar characters. This characteristic enables higher compression ratios when using techniques such as run-length encoding. The transformation is reversible, meaning it is possible to produce the original string from the Burrows-Wheeler transformed string, without any other information. Sorting S would enable similar, or possibly better, compression ratios using run-length encoding, but it will not be reversible.

A string S of n characters is transformed by the Burrows-Wheeler transformation [4, Section 2] by forming n cyclic shifts of S. These n permutations of S are then sorted in lexicographical order. An extra character (#), not in the alphabet of S, is added to keep track of the end of the original string. The BWT of S is then the concatenation of the last character of each permutation in sorted order, excluding #.

In Figure 4 we present an example transformation of the string *bananahat*. The list to the left in Figure 4 is the cyclically shifted permutations of S and the list to the right contains the same permutations, but in lexicographically sorted order. The result of the Burrows-Wheeler transformation is then the characters at the last index in each column, highlighted in bold in Figure 4. The Burrows-Wheeler transformation of $S = bananahat$ becomes BWT(S)= $tnnbhaaaa$. The original string is identified by having a # at the end.

Looking at BWT(S) we can see that equal characters are now grouped together. It would not make much sense to compress something without being able to decompress it again. Burrows et al. [4, Section 2] describes an algorithm for getting the original string from the Burrows-Wheeler transformed string. Their algorithm is not very intuitive, so

$$M = \begin{bmatrix} dca\# \\ ca\#d \\ a\#dc \\ \#dca \end{bmatrix} \Rightarrow M' = \begin{bmatrix} \#dc\mathbf{a} \\ a\#d\mathbf{c} \\ ca\#\mathbf{d} \\ dca\mathbf{\#} \end{bmatrix}$$

Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4
a	#	*a#*	*#d*	*a#d*	*#dc*	*a#dc*	*#dca*
c	*a*	*ca*	*a#*	*ca#*	*a#d*	*ca#d*	*a#dc*
d	*c*	*dc*	*ca*	*dca*	*ca#*	*dca#*	*ca#d*
#	*d*	*#d*	*dc*	*#dc*	*dca*	*#dca*	**dca#**

Figure 5: Example of how to do reverse BWT on string "*acd#*". The returned value is "*dca#*".

we have added a description of a more intuitive algorithm [2] that reverses BWT. It is worth noting that the algorithm we describe is less efficient than the one Burrows et al. [4, Section 2] describes. The point of describing a more intuitive algorithm is to more easily convince the reader that it is possible to reverse BWT.

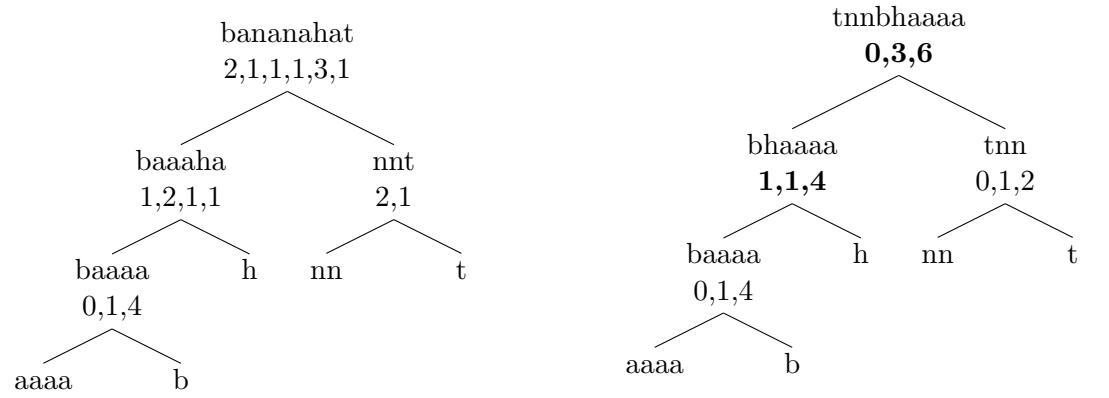
It is possible to reverse BWT by taking the BWT, sorting it and then adding the BWT in front of the sorted value and then sorting that. This procedure continues until the number of characters in each row is equal to the length of the BWT. An example of the process in shown in Figure 5. After each sorting step each column in the sorted result corresponds to the column at the same position in *M'*. After the last sorting the result is equal to *M'*. The original string is then the value with the end of line character # at the end.

Figure 6 shows two small examples of wavelet trees using run-length encoding, one constructed on the string *bananahat* (Figure 6a) and the other on the Burrows-Wheeler Transformation of *bananahat* (Figure 6b). We can see in Figure 6b, highlighted by the numbers in bold, that fewer Run-length encoded values needs to be stored than for the non-Burrows-Wheeler transformed string in Figure 6a.

One might wonder how RANK and SELECT queries can be useful when the input string is Burrows-Wheeler transformed, as the results of the queries become essentially unrelated to the original string S, without an obvious way of transforming the query results back to what they would have been on a tree constructed on the original non-transformed string. We have not found any way to transform the results back, and the main use of constructing a Wavelet Tree on the BWT of a string seems to us, by far, to be compression.

Rank and select queries on the BWT of a string does, however, have uses when working with the *FM-index*, named after its inventors Paolo Ferragina and Giovanni Manzini, which is a self-index based on the Burrows-Wheeler transformation $BWT(S)$ that is able to find occurrences and positions of patterns (sub-strings) in S by looking at $BWT(S)$. The procedure for doing so is described by Mäkinen and Navarro [6, Section

[2] `http://en.wikipedia.org/wiki/Burrows-Wheeler_transform#Explanation`



(a) RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$

(b) RLE Wavelet Tree on $BWT(bananahat) = tnnbhaaaa$ with alphabet $\Sigma = abhnt$

Figure 6: Comparison of Wavelet Trees using Run-Length Encoding on a string and its Burrows-Wheeler Transformation

2].

4.2.4 Huffman-shaped Wavelet Trees

Mäkinen and Navarro [6, Section 4] describes a Huffman Shaped Wavelet Tree which skews the tree to one side and places symbols with higher frequencies towards the other side so that they are closer to the root than those that have a lower frequency. More precisely, they are placed in the tree in such a way that the path from the root to a leaf corresponds to the binary Huffman Code [16, Introduction] of the symbol of that leaf. Using a Huffman Shaped Wavelet Tree is an alternative to run-length encoding.

This approach skews the tree and as a result increases the height of the tree, which for uniform data would result in higher average query time, but by placing the most frequent symbols highest and least frequent symbols lowest, it decreases query time massively for symbols with high frequency. Queries on a Huffman shaped Wavelet Tree for a symbol that has a high frequency then returns faster than queries for a symbol that was less frequent. Assuming symbols that occur with high frequency are also queried for more often, the average query time are reduced when using a Huffman-shaped wavelet tree.

The Huffman Code [16, Introduction] of a symbol occurring with high frequency is a shorter binary string than the Huffman code of a symbol occurring with low frequency. The most frequent symbol could be encoded in as little as one bit! This entails that the storage space required for the many occurrences of the most frequent symbols would be massively reduced, while the space required for the least frequent symbols would be increased. If the difference in frequency is sufficiently high, the reduction in space for the most frequent symbols would outweigh the increase in space for the least frequent and the overall storage requirement would be reduced.

Since the Huffman encoding is based on frequency of symbols it achieves the best

performance and space complexity when symbols are non-uniformly distributed. If the data is uniformly distributed then the length of all Huffman codes would be similar resulting in a balanced tree having performance and space complexity similar to a normal Wavelet Tree.

4.3 Information Retrieval

A wavelet tree can be used to efficiently answer numerous queries in different problem domains. In this section we describe in some detail a select number of information retrieval scenarios.

4.3.1 Access, Rank, and Select Queries

The three queries supported by a wavelet tree are access, rank, and select. They are often used together to answer more complex queries when the wavelet tree is used as e.g. a dictionary or a self-index. They can also form the building blocks for many other, more advanced algorithms and queries.

The access query for position p will return $S[p] = c$, or, the character c at position p in the original input string S. The wavelet tree supports access in $O(\log \sigma)$ time.

The rank query for a character c and a position p will return how many times the symbol s occurs in the input string up to position p. The select query for character c and occurrence parameter o will return the position of the oth occurrence of c in the input string.

G. Navarro [1, Section 5] points to an application found by Ferragina and Manzini [17, Section 3] that uses access and rank[3] queries to find the number of occurrences of a pattern p in a string S by storing and querying the Burrows-Wheeler transformation of the string S^{BWT}, enabling compression along with efficient query times. G. Navarro [1, Section 5] also point to other similar results and improvements on the previous results by others, showing there is a wide interest in using wavelet trees to store a sequence and query for the occurrences of patterns within that sequence.

G. Navarro [1, Section 5] further points out the uses of a wavelet tree as a positional inverted index. By storing the list of word identifiers in the wavelet tree both the text itself and the inverted index is stored. Access queries will then return the word at the given position while $\text{select}_c(S, o)$ can be used to get the oth word in the inverted list of a word c for the string S. Rank queries can be used effectively in some list intersection algorithms. The efficiency can be improved by using multi-ary wavelet trees or Huffman-shaped wavelet trees as the non-uniformity of word usage in language makes it a good candidate for Huffman coding.

The positional inverted index application can also be extended to *document retrieval* [1, Section 5] by introducing a document boundary character such as \$ and storing the concatenation of all the documents with the document boundary character in between each. The first document containing some word c is document number $j = \textsc{Rank}_{\$}(S, p) + 1$ where $p = \textsc{Select}_c(S, 1)$. Document j ends at position

[3] Ferragina and Manzini calls rank queries "Occ" in their paper

$p' = \text{SELECT}_\$(S, j)$ and contains $o = \text{RANK}_c(S, p') - \text{RANK}_c(S, p)$ occurrences of the word c. The next occurrence of the word c in another document is at $p_{next} = \text{SELECT}_c(S, o{+}1)$.

4.3.2 Range Quantile Query

A range quantile query is a query that returns the kth smallest number within a subsequence of a given sequence of elements. If we are e.g. given a list of price changes on a laptop during the last year then a range quantile query is able to answer what the kth-smallest price of the laptop was within for instance a month of that year. It is therefore also easy to find quantiles like the 2-quantile (median) or the 3-quantile. To e.g. find the median, k can be defined as half the length of the subsequence. This would return the middle element of the subsequence. The 3-Quantile can be found by setting k to $\frac{1}{3}$ of the length of the subsequence. Quantiles are important values within such fields as statistics and economics.

Range Quantile queries especially are interesting to us because they do not require any changes to the wavelet tree and uses it in its simple form. Our optimizations can therefore be applied directly without modification.

Gagie et al. [8] show how the wavelet tree can be used to support efficient range quantile queries on a sequence S of n numbers in $O(\log \sigma)$ time if $rank_b$ is supported in $O(1)$ time [8, Section 3]. The range is denoted as $S[l..r]$. A range quantile query based on a wavelet tree works by computing two rank queries on the bitmap of each node in a traversal from the root to a leaf node.

Algorithm 4 Range Quantile Query

```
function RangeQuantileQuery(k, l, r)
    if current node is leaf then
        return number in leaf
    end if
    0sInRange ← rank_0(S[l..r]) = rank_0(r) − rank_0(l − 1)
    if 0sInRange ≤ k then
        l = rank_0(l − 1) + 1
        r = rank_0(r)
        return LeftNode.RangeQuantileQuery(k, l, r)
    else
        k = k − 0sInRange
        l = rank_1(l − 1) + 1
        r = rank_1(r)
        return RightNode.RangeQuantileQuery(k, l, r)
    end if
end function
```

The two queries are $rank_b(l-1)$ and $rank_b(r)$ where $rank_b$ is the binary rank. $rank_b(l-1)$ is used to find the number of 1s and 0s in $b[1..(l-1)]$ and $rank_b(r) - rank_b(l-1)$ gives the number 1s and 0s in $b[l..r]$. The algorithm goes to the left if there

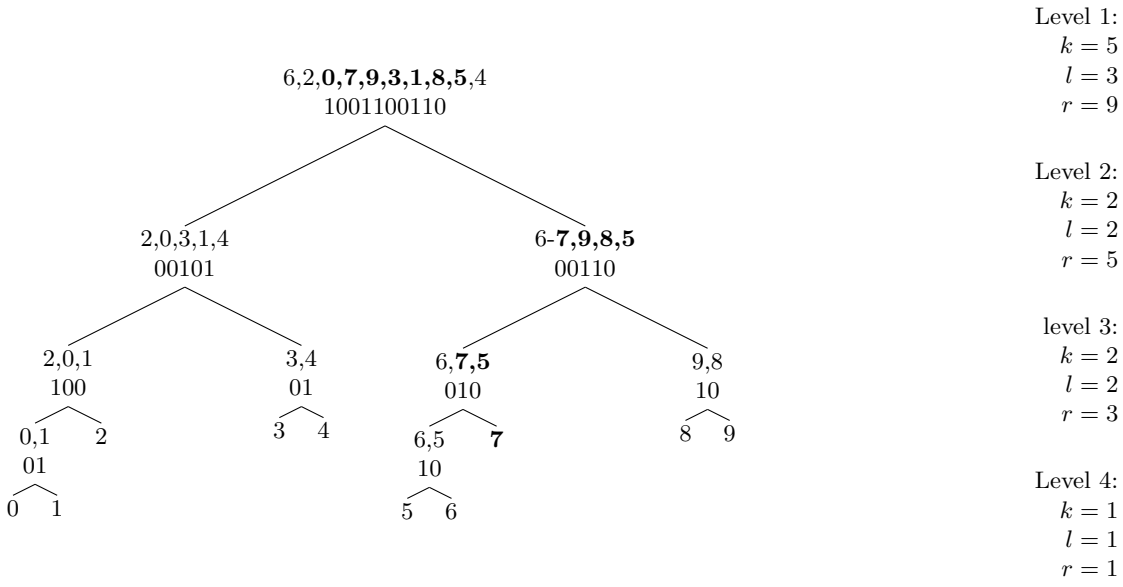


Figure 7: Range Quantile Query on a Wavelet Tree. $S = \{6, 2, 0, 7, 9, 3, 1, 8, 5, 4\}, k = 5, l = 3, r = 9$.

are more than k 0s in $b[l..r]$ and set $l =$ (number of 0s in $b[1..(l-1)]$)$+1$ and $r =$ (number of 0s in $b[1..r]$). The algorithm goes to the right if there are less than k 0s in $b[l..r]$ and subtract the number of 0s in $b[l..r]$ from k and set $l =$ (number of 1s in $b[1..(l-1)]) + 1$ and set $r =$ (number of 1s in $b[1..r]$). This procedure continues recursively until it hits a leaf and then returns the number stored in the leaf which corresponds to the kth smallest number in $S[l..r]$.

Algorithm 4 describes the pseudo-code for a range quantile query where $rank_1$ is a binary rank query which returns the number of 1s in the bitmap of S in each node generated by the wavelet tree. $rank_0$ return the number of 0s. The argument to $rank_0$ and $rank_1$ is the position to find the number of occurrences up to. This means that $rank_1(r)$ for instance returns the number of 1s in the bitmap until position r.

An example of a range quantile query can be seen in Figure 7. The numbers in bold indicates the range $S[l..r]$ where $S = \{6, 2, 0, 7, 9, 3, 1, 8, 5, 4\}$ and $l = 3$, $r = 9$ and $k = 5$. When $k = 5$ it means that we are looking for the 5th smallest number within $S[l...r]$ which is 7 indicated by the leaf that the query ends up in before terminating. l and r indicates the range to look within. The right side of the figure shows how k, l and r develops in each recursive call of the Range Quantile Query.

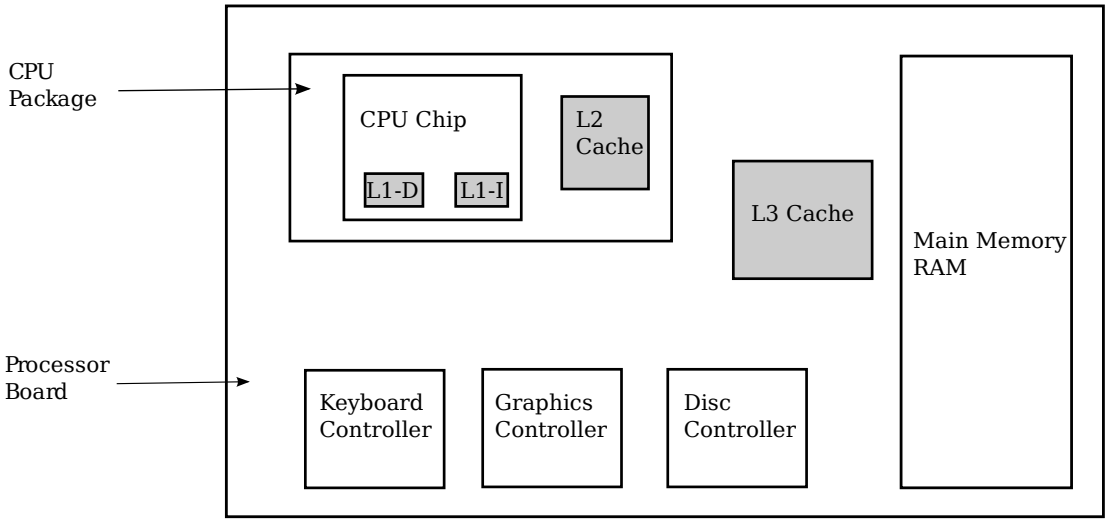


Figure 8: The three cache levels. Figure borrowed from: [11, Section 4.5.1]

Part II
Hardware, Implementation & Test

In this part of the thesis we describe what hardware penalties we focus on optimizing and how and why they occur. We discuss choices we have made in the implementation like which C++ data structures to use. We also describe our test setup, what tools we have used for testing and discuss the effect of using uniform vs. non-uniform distributed data in our tests.

5 Cache, Branch Prediction and Translation Lookaside Buffer

In our optimizations we try to make our algorithms better utilize the way certain hardware components function on the computer in order to improve the practical running time. These components are the cache, the branch prediction unit and the translation lookaside buffer.

5.1 Cache design and cache misses

A cache is a small fast memory storage that holds the most recently used memory. Using caches can improve the memory access time many-fold, which is important since memory access is often a bottleneck in programs. Modern CPUs have a prefetcher that attempts to predict future memory accesses and fetches them into the cache ahead of time. Therefore, the pattern in which a program accesses the memory can have significant influence on the performance of the program. Modern memory systems usually have three levels of caches: Level 1 (L1), Level 2 (L2) and Level 3 (L3) [11, Section 4.5.1].

Figure 8 shows how the three cache are placed in the relation to the CPU. The L1 cache resides on the CPU chip itself and usually has a size in the range from 16 KB to 128 KB and because it is placed directly on the CPU chip it is able to provide extremely fast memory access. The L2 cache is placed next to the CPU chip on the CPU package and it is connected to the CPU via a high speed path. The L2 cache typically has a size between 512 KB and 1 MB which means that it can hold more data but is not able to provide as fast access as L1. The L3 cache is placed on the processor board and usually has a size around 3 MB. Since it is placed further away from the cpu it is not able to provide as fast access as L1 and L2 but still much faster than fetching data from RAM.

All three caches are inclusive which means that L2 contains the data from L1 and L3 contains the data from L1 and L2. This means that if data is evicted from L1 it will still reside in the L2 and L3 caches or if data is evicted from L2 it will still reside in L3. This is an advantage because it allows fast access to data even if it is evicted from L1 or L2. If the caches were not inclusive then it would result in a many more data requests to the main memory.

There are two types of address locality that the caches exploit to improve performance: Spatial Locality and Temporal Locality. Spatial Locality is that, when memory locations have addresses numerically close to a recently accessed memory location, they have increased probability of being accessed soon. Temporal Locality is that, when memory locations have been accessed recently, they have increased probability of being accessed again soon. The cache exploits spatial locality by fetching more data than has been requested, either by loading a larger chunk of memory than requested, such as a cache-line, or by speculatively fetching more cache-lines based on previously recognized access patterns. This speculation is performed by the cache prefetcher and assumes that it is possible to anticipate future requests by looking at the previous access pattern. Temporal locality is exploited by choosing what to evict on a cache miss and normally it is the data entries that has been accessed least recently that are evicted.

Data in the main memory is split into blocks of fixed size called *cache-lines*. There are usually 4 to 64 consecutive bytes in a cache-line. Some of these cache-lines are always present in the caches. If a requested word is in the cache, a trip to main memory can be avoided, but if the word is not in the cache then a cache-line must be evicted from the cache (assuming that the cache is full, which it always is after the first few seconds of operation) and the cache-line containing the word must be fetched from main memory or a lower level cache if one is present. This is called a *cache miss* and has a high penalty because fetching a new cache-line is expensive. The general idea is to have the most heavily used cache-lines in the caches as much of the time as possible to reduce the amount of cache misses.

5.1.1 Cache associativity

When designing a cache it is important to consider whether each cache-line can be stored in any cache slot or only in some of them. There are three approaches to solving this problem; *direct mapped cache*, *n-way set-associative cache* and *fully associative cache*.

In a *direct mapped cache* each cache-line can only be stored in a specific cache slot,

the address of which is found by using a mapping function on the address of the original main memory address, e.g. the address modulo the number of cache slots. This means that two cache-lines cannot be mapped to the same slot simultaneously. Given a memory address it is only necessary to look for it in one place in the cache and if it is not there then it is not in the cache. Using this approach, and an appropriate mapping function, consecutive memory lines are placed in consecutive cache slots. The problem with a *direct mapped cache* is that since there are many more cache-lines in main memory than there are space for in the cache, many cache-lines ends up competing for the same slot. These competing cache-lines might end up constantly evicting each other which results in a substantial performance loss.

This problem can be fixed by using a *n-way set-associative cache*, which is a cache that allows n slots for each cache address. This way if we have two cache-lines A and B whose addresses map to the same cache address and that address is already occupied by A while B tries to use the same address then A does not have to be evicted because the cache has $n - 1$ other slots to place B in. If all n slots are occupied, then a cache-line from one of them has to be evicted. The question then becomes: Which one?

A popular algorithm, that determines which cache-lines to evict is called *LRU (Least Recently Used)*. It works by keeping an ordering of the slots at a cache address. When a cache-line that is present in the cache is accessed the LRU algorithm updates the list by moving the entry corresponding to the accessed cache-line to the top of the list. When an entry needs to be replaced it is the one at the end of the list that is evicted because it is the least recently used entry.

A *fully associative cache* allows any cache-line to be saved in any cache slot but it is complicated and costly to implement in hardware because it for instance might need to keep an ordered LRU list for the entire cache which would require a lot of bookkeeping.

The *n-way set-associative cache* is the most popular choice because it has a good trade-off between implementation complexity and cache-hit rate.

5.2 Branch Prediction and Misprediction

There are many steps in executing a single instruction in a modern computer. It has to be fetched, decoded, registers has to be loaded with the required data etc. Because of this, modern computers are highly pipelined, meaning that they execute different steps for consecutive instructions in parallel. That is, instruction 1 might be executed while instruction 2 is being decoded while instruction 3 is being fetched from the program code memory section. A pipelined architecture can cause great speed improvement, but because of how it works, it works best on non-branching code because the result of a branch determines which instructions should be fetched, decoded and executed next. When a branching instruction is encountered, the CPU can either choose to stall until the branching instruction has been executed, or it can try to predict the outcome of the branch before it is executed with the condition that it must be able to roll back any instructions executed between the prediction and the execution of the branching code. Modern programs are typically full of branch instructions.

if $i = 0$ **then**
    $k \leftarrow 1$
else
    $k \leftarrow 2$
end if

A possible translation to assembly looks like this:

```
1.              CMP 0, 1        : compare i to 0
2.              BNE Else        : branch to Else if not equal
3. Then:        MOV k, 1        : move 1 to k
4.              BR Next         : unconditional branch to Next
5. Else:        MOV k, 2        : move 2 to k
6. Next:
```

Figure 9: Program fragment with conditional and unconditional branches

There are conditional branches and unconditional branches. An example of an unconditional branch and a conditional branch is shown in Figure 9 [4] where *BNE Else* is a conditional branch and *BR Next* is an unconditional branch.

An unconditional branch is a simple jump to a specified label, it is not based on a condition and is less of a problem than conditional branches because while the target of the jump is known before the instruction is executed, it is not yet known when the fetch unit goes to fetch the next instruction. Having no branching instructions also means the fetch unit is able to read consecutive words from memory and make better use of prefetching.

A conditional branch jumps to one of two places in the code based on whether a given condition is true or false. The ambiguity in a conditional branch is problematic because of the nature of modern pipelined CPU architectures where the stage of the pipeline that computes the result of a comparison is many stages later than the fetching unit. Before the result of the comparison is computed, the fetcher does not know where in the program code to fetch from.

Old pipelined machines would just stall until it was decided what branch to take. Doing this has a heavy impact on performance, especially if there is a lot of conditional branches in a program. Modern machines try to predict what branch will be taken, using a *branch prediction unit*, and then executes that code until it is known whether the branch was predicted correctly or not. If the branch was predicted correctly then the execution simply continues. If the branch was mispredicted then the executed instructions in the mispredicted branch needs to be rolled back and the correct branch must be taken. Undoing the effects of the wrong execution path is an expensive operation which means that it is important to minimize the amount of branch mispredictions as much as possible

[4] Example borrowed from: [11, Section 4.5.2]

to get good performance.

5.2.1 Branch Prediction techniques

There are generally two ways to do branch prediction; Static branch prediction and Dynamic branch prediction.

In **Static branch prediction** the branch taken is always fixed. There are, in general, four ways to choose what branch to take and it is the compiler that chooses which to use of the first three based on where it makes sense. The profile-driven prediction scheme is not something that the compiler can suddenly decide to do:

1. **Branch always not taken:** It is assumed that the branch is not taken which means that the instruction flow can continue as if the branch condition is false.

2. **Branch always taken:** This works in the opposite way of the above. Here it is assumed that the condition is always true and the branch is taken. This approach makes sense with a pipeline where the branch target address is known before the branch outcome.

3. **Backward taken forward not taken:** Here backward branches are always taken which for instance is the branch at the end of a loop that go back to the beginning of the next loop iteration. The forward branches are not taken.

4. **Profile-driven prediction:** Here the branches of the program is profiled by running the program and the information is given to the compiler to use for branch prediction. This of course requires the program to be compiled then run in order to be profiled and then compiled again to incorporate the branch info.

In **Dynamic branch prediction** the branch prediction is carried out at runtime and tries to the adapt to the program's current behaviour. This is better than just having some static schemes to choose from because it allows more complex and usually more correct decisions. The basic idea in dynamic branch prediction is to use the past branch behaviour to predict the future branch.

One way to do dynamic branch prediction is to have a history table that logs conditional branches as they occur, to be able to look up what direction they took when they appear again. The simplest way to implement the history table is to have it contain one bit for each conditional branch instruction that indicates whether the conditional branch was taken or not last time it was executed. With this approach the branch will be predicted to go in that same direction as it did the last time. If the prediction is wrong the bit in the history table is flipped.

Using only one bit in the history table to indicate that a branch was taken or not poses some problems. When a loop is finished the branch at the end will always be mispredicted and change the bit in the history table. When the loop is run again the branch at the end of the first iteration will be mispredicted. In the case of nested loops occurring in a frequently called function, the amount of mispredictions increases and the performance suffers.

To eliminate the loop mispredictions two bits can be used instead of one in such a way that a branch must be predicted wrong twice in a row for the prediction scheme to change. This means that the different possible bit values becomes 00, 01, 10 and 11. Bit value 00 indicates that the last two branching instructions resulted in not jumping and "no jump" is predicted. If a conditional branching instruction is reached where this prediction is wrong the last bit is set to 1 and the bit value becomes 01. The predicter still predicts "no jump". If this prediction is correct for the next conditional branching instruction the binary value is set back to 00 and the prediction continues predicting "no jump" as before. If the previous "no jump" prediction instead was wrong at a 01 value, it will be changed to 11 and the future prediction changed to "jump". Two mispredictions are then required to change back to bit value 00 and "no jump" prediction, in a similar way it goes from 00 to 11, except using 10 as the in-between value.

When the branch is correctly predicted there is still a problem. The address to go to for some conditional branches are computed values obtained from doing arithmetic on registers. Since computation takes place after fetching, the address is unknown and the prediction becomes useless. A way to fix this problem is to store the address branched to last time for the particular branch, in the history table. The previous address can then be used to branch to when the corresponding conditional branch is predicted again.

5.3 Virtual Memory and Translation Lookaside Buffer misses

Modern computers use virtual memory to address the problems of sharing limited memory between multiple processes and/or users. Virtual memory hides the presence of physical memory and instead presents an abstract view of main memory by concealing the fact that physical memory is not allocated to a program as a single continuous region while also concealing the actual size of the main memory. This creates the illusion that the available memory for a given process is larger than what is physically available. The illusion is accomplished by dividing the virtual memory into smaller subsections called **pages**, which can be loaded into physical memory when needed. Translating a virtual address into a physical address is an expensive operation and the Translation Lookaside Buffer (TLB) helps to speed up this process.

5.3.1 Virtual memory: Pages

A virtual memory address *va* is interpreted as a pair containing the page number and a word number (offset within the page). A physical memory address *pa* is interpreted a pair consisting of the page frame number and the word number within the frame [12, Section 8.2.1].

When loading a virtual memory page into the physical memory it is necessary to be able to translate *va* to *pa*. Since the word number is the same in both *va* and *pa* the translation only needs to find the physical page frame that contains the virtual page. Therefore it is necessary to keep track of the page and its current page frame which can be done using a page table.

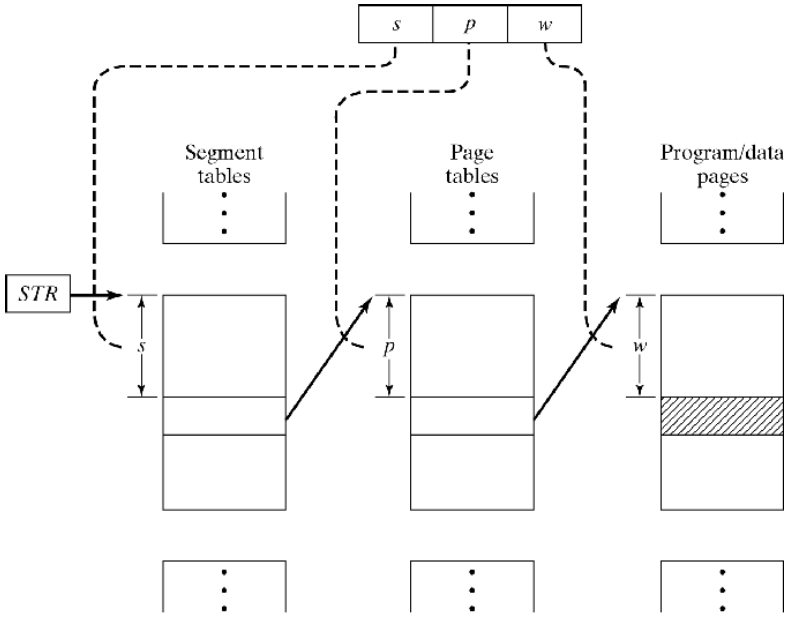


Figure 10: Virtual Address Translation. This figure is borrowed from [12, Figure 8-7]

5.3.2 Virtual memory: Segmentation

Sometimes a process contains multiple dynamically changing elements. Placing such elements in a single address space is a difficult problem. The problem is solved by segmentation [12, Section 8.2.2].

A segment is a collection of address spaces that can have different sizes. This allows the virtual memory to be organized in the same way as a given application by using a segment for each logical element in the application e.g. a function, an array or table. Segmentation and paging is combined to allow a multisegment address space while also having a simple address translation algorithm. The concept is shown in Figure 10.

5.3.3 Translation Lookaside Buffer

The translation from *va* to *pa* gets an extra parameter when adding segmentation; the segmentation number. The physical address is then found by first finding the page in the segmentation table and then finding the corresponding frame in the page table and then finding the frame in the physical memory. This means that the translation from *va* to *pa* requires three physical reads.

To reduce the number of reads needed when a virtual address is translated, a translation lookaside buffer (TLB) is used [12, Section 8.2.5]. The TLB saves the most recent translations of *va* to *pa*, i.e. translated page numbers, to make them quickly available for future use. This means that subsequent accesses of a virtual address within a short period is able to bypass segmentation- and page table lookups and simply just access the needed page frame from the TLB and find it in the physical memory. When the page

is available in the TLB it is called a *TLB hit*. The TLB is not a large buffer and it can only hold a few virtual page translations at a time. A *TLB miss* occurs when a given virtual page translation is not available in the TLB and as a consequence the virtual page needs to be translated using table lookups. A TLB miss is therefore expensive and it is a good idea to minimize them as much as possible to improve performance.

6 Notes on Implementation

6.1 Using Integers as Characters

The Wavelet Tree is a data structure for strings. Using the C++ `char array` or C++11 `string` types would seem natural in this case, but they each have problems. The C and C++ `char` type is only of size 1 byte allowing us only to use an alphabet size of up to 256. This makes testing the dependency of the running times on alphabet size difficult, as we expect inaccuracies in the running time will exceed the difference in running time between the available sizes of the alphabet.

The C++11 `string` and arrays of type `char32_t` does not have this problem and supports character types up to 32-bit unsigned. The problem then lies in output and readability as characters corresponding to byte values below 32 are special non-printable control characters such as carriage-return and backspace. At higher byte values other non-printable control characters and otherwise unreadable characters appear again. This means we would have to be selective with the allowed byte values in our alphabets if we want it to be readable for output and debugging, thereby ending up with an alphabet that is non-continuous on the set of byte values as a result, which is inconvenient. Because of this, we have for convenience chosen to simply use vectors of integers as our strings in our implementations. E.g. we will use `uint` instead of `char32_t`, which both take 4 bytes of memory. We expect that this will have no impact on performance as both characters and integers are simply different representations of byte values.

In our implementation, we assume that the alphabet is always continuous on the sorted set of byte values, i.e. the alphabet spans all values possible between some minimum and maximum value, with no gaps. Thus, we store the alphabet as a minimum and maximum value, instead of storing each value in some data structure to pass around or point into. This is for convenience as any other non-continuous alphabet could simply be mapped to a continuous run of byte values and used in the same way. This mapping could e.g. be done by storing an array of the alphabet in sorted order and using pointers into this array to signify the characters. Lookup into the array is not necessary unless printing for human reading, since comparison of the pointer addresses returns the same result as comparing the bytes.

We will still use the terms “character/symbol” and “string” in our descriptions of the algorithms even though we have implemented them as integers and integer arrays, as we feel the terms “character/symbol” and “string” are more intuitive and give clarity.

6.2 Generating the Data

We implemented a small script in Python to generate our input strings of 4-byte integer values and write them in binary format to files. This was slower than e.g. piping from `/dev/random` into a file, but we needed to constrain the alphabet and even though slow, a script was the easiest way to achieve that.

6.3 Reading Input

At first the input data was read from stdin using the `getline(cin, &string)` function. Once we applied a profiler we found this to be horrendously slow, our Naïve algorithm spending about 20 % of its running time on resizing IO buffers. We then switched to using the `ifstream` class and IO time was reduced significantly to below 1 % of total running time.

6.4 Verifying the Results

To ensure that our implementations are correct, we implemented some simple and slow algorithms in python to calculate rank and select on the same input data we construct the wavelet trees on. The point being that the python implementation should be so simple and easy to understand that it cannot contain errors and therefore produce the correct results for comparison. We then compare results from rank and select queries on our wavelet tree to results from the same queries using the python implementation. When they agree on several randomly selected sets of query parameters, we feel confident that our wavelet tree construction, rank, and select implementations are correct.

6.5 Combating Over-Optimization

The C++ compiler (g++) in the GNU Compiler Collection (GCC) is an optimizing compiler and can sometimes using static analysis recognize that the results and possible side-effects of a computation will not be used in the code and will in those cases completely remove that computation from the compiled code as an optimization. This means that the compiler could potentially remove the parts of or the entire computation for our queries when we test them, if the results are not used for anything. To ensure that the compiler does not throw needed computations out the window in our tests, the results of each query is collected in an array and printed to stdout. It is only printed after the collection of measurements is done to affect the running time minimally.

6.6 Reducing Construction Time Memory Usage

Since the Wavelet tree is a recursively defined data structure, we also implement it recursively. This causes any stack-allocated variables to be held in memory until we leave the scope of the constructor function. We traverse and split the input string into its left and right parts in each node constructor and thus end up holding the input string twice in memory: once in the variable holding the input string and once in the

two variables holding the left and right split strings. This is wasted memory because the input string is not actually needed any longer once we have split it into its left and right parts. Because one sub-node constructor is simply called first and then the other when the first has completed and finally return once both subnodes has completed constructing themselves, we end up completing the construction of the nodes in post-order. This means the scopes of the root node and those near the root is kept alive for most of the running time of the construction algorithm, and much memory is wasted. The solution is to allocate these strings on the heap instead, passing pointers to the subnode constructors and having them delete them (as their input strings) once they have split them. Doing this reduced the memory usage so much that we could run it for input strings with a length above 10^8 characters without exhausting the 8GB available memory on our test machine.

6.7 Bitmap implementation choice

There are several bitmap implementations available to us. In the Standard Templating Libary (STL) of C++ there is `std::bitset<size_t N>` and `std::vector<bool>`. From the Boost library there is `boost::dynamic_bitset<>`.

`std::bitset` While it would technically be possible to use the `std::bitset`, it requires that the size of the bitset is known at compile time and passed as a template parameter. This means it would be necessary to recompile the program for each size n of the input string. It would also be necessary to allocate a bitmap with room for n bits for each sub-node as that is the theoretically possible size required, making the size required for the bitmaps of the tree $O(n \cdot |nodes|) = O(n \cdot \sigma)$ instead of $O(n \cdot h) = O(n \log \sigma)$. Another reason why we cannot use `std::bitset` is because it does not support pointer access, which means that it is impossible to do queries using `popcount`, which is a CPU instruction we utilize to improve the practical running time of Rank and Select queries and is described in Section 8.1. We also expect that an actual usable practical implementation should be able to handle different sizes of input at run time instead of compile time.

`std::vector<bool>` is a specialised implementation for `bool` that packs the data so that each `bool` only takes up one bit and is not an actual C++ container, though it tries to mimic some of the behaviour. It is basically the STL implementation of a dynamically allocated bitset. This is the implementation we decided to use for our bitmap because it allows dynamic allocation and pointer access.

`boost::dynamic_bitset` is the Boost library's take on a dynamic bitset. It does not try to mimic a container and lacks some features such as an iterator because of that. It also does not guarantee that the bits will be allocated consecutively in memory and has no raw pointer access to the data in memory. This is a problem when calling popcount on all machine words from beginning up to some index.

We chose to use `vector<bool>` mainly because it supported direct pointer access into the backing array and that backing array was a single continuous array so we could do

pointer arithmetic across an entire bitmap. `boost::dynamic_bitset` does not support any of this.

Pieterse et al. [18] has tested 5 different bitvector implementations for C++, including `std::vector<bool>` and `boost::dynamic_bitset`. In terms of running time, `std::vector<bool>` performs the worst of the tested bitvectors for their test case. But, their test case does not at all resemble the way we use it. We do not utilize any of the extra functions or features they have tested such as the reset operation and bitwise operations on entire arrays.

In terms of memory, `std::vector<bool>` performed the best by using the least amount of memory, owing to its simple implementation storing no additional meta information and using only a single raw array as its backing data format. This is a characteristic we like for our purposes as we basically use it as an array supporting access to single bit values.

6.8 Challenges in Implementation

The wavelet tree is a somewhat simple data structure as a tree structure of bitmaps implemented using pointers and dynamic bitsets. The construction of the wavelet tree was not a great challenge, neither was the basic forms of rank and select queries. But, in later iterations of our wavelet tree, we implemented more and more intricate designs of the bitmaps, spending more and more time debugging to make it work absolutely correctly.

We are not experts in C++ and have in fact programmed very little in it previously, which both introduced and complicated many issues that someone with more experience with C++ would likely have had little trouble with. The sheer number of different algorithms we implemented for the various variations of the wavelet tree only exacerbated the time spent implementing and debugging.

Notable challenges include implementing rank and select queries that utilized concatenated bitmaps, implementing support for aligning the precomputed blocks with machine pages, and handling and masking machine word correctly when using the cpu intrinsic popcount instruction.

Popcount Instruction works on whole machine words at a time and so our code had to figure out when it was worth using the instruction and handle any excess bits counted or any bits not counted.

Concatenated Bitmaps required that our code correctly handled the edge cases where bitmaps touch each other, making sure to only count the number of 0s or 1s on the correct side of the boundary. This was done by using pointer arithmetic and calculating various offsets, misalignments and offsets of offsets and misalignments. It only became more complicated when using precomputed values as well.

Page-aligned Blocks only introduced more handling and bookkeeping of offsets and offsets of offsets.

7 Notes on The Experiments

Here we discuss some things general for all our experiments, or all those where applicable.

7.1 Testing Machine Specifications

CPU	Intel Core i5-3230M
OS	Ubuntu 14.04 64-bit
Kernel	Linux 3.13.0
RAM	8 GB
Level 1 Data Cache	32 kB
Level 2 Total Cache	256 kB
Level 3 Total Cache	3072 kB

7.2 General Setup

Our code was compiled using GCC 4.8.2 with compiler flags `-O3 -std=c++11 -march=native`. The `-march=native` flag was necessary to use the native `popcount` cpu instruction. Our PAPI library version was 5.4.0.0 using perf version 3.13.11-ckt18.

We ran 1000 queries 5 times for each variable parameter and registered the total running time for each set of 1000 queries and then used the average of those 5 as the result. Examples of variable parameters used in our tests are: the alphabet size, the block size and the skew of the tree.

We calculated the standard deviation of the 5 runs and include it in the graphs as errorbars. All our graphs include the standard deviation as errorbars. If one appears not to have any errorbars that means the standard deviation is so small it is difficult or impossible to see.

7.3 Choice of Input String

We have chosen to construct the input strings used in our experiments so that each character occurs with the same probability at each position. This means the string has a uniform distribution of characters from the alphabet. We have chosen to do so for several reasons, among them being that we think it is a realistic use case, e.g. for Range Quantile Queries or Geometry Processing, as well as making the choice of character to query for in our experiments make less difference. The even amount of occurrences of each character also means there will be little difference in the size of bitmaps between the nodes in a single layer of the tree.

7.3.1 Uniform vs. Non-Uniform data

Uniform data for a wavelet tree is a string with each character occurring with the same probability at each position and therefore with similar frequencies. If the data is non-uniform it means that some symbols from the alphabet will appear with a significantly higher frequency than others. If one knows the frequencies of all symbols in the alphabet,

without needing it to be exact, then one can build a Huffman shaped Wavelet Tree (described in Section 4.2.4) and we expect it will beat out any balanced wavelet tree in terms of performance. The frequencies can be found by a simple linear scan of the input string before building the tree.

We are more interested in the general case of being able to perform well on any input string and so we do not want to implement optimizations that require a specific character distribution in the input string. Using non-uniform data for our testing with a general wavelet tree will also introduce bias into our results. This is because the sizes of the bitmaps in each node in a given level of the tree would not be equal, as they would for uniform data. When we query the tree and take a path with many large bitmaps it will take longer than a path with many small bitmaps. Depending on which non-uniform distribution is used, some characters might not even appear at all in the input string, and querying for them would terminate early. This is especially true for Select queries that find the leaf node corresponding to the character that was queried for and then spends most of the its time traversing up the tree looking for the position in the complete input string. If the queried-for character did not exist in the input string, a select query would terminate before even having gone down the full length of the tree, because the leaf node corresponding to the character does not exist. Rank queries on the other hand would still take much the same time on non-occurring characters as on occurring characters, because they spend most of their time in the single downward traversal of the tree they perform, and still will perform for non-occurring characters as it is only near the end of this traversal that the character will be found to not occur.

Therefore having non-uniform data would introduce a bias in our query tests based on the symbol we are querying for and it is a bias that would be difficult to avoid without introducing more bias by choosing exactly which characters to query for.

There is also the problem of choosing which occurrence to query for in the case of select, as the character should occur at least that many times. When using uniform data we know that it is extremely unlikely for any character to occur less than some minimum number of times, because of their equal occurrence probability.

If we used non-uniform data we would also have the problem of choosing which non-uniform distribution we should use, and there are many to choose from.

To compare the effects of using uniform vs. non-uniform data we have made an experiment that compares the running time of building the wavelet tree and doing Rank and Select queries for the two distributions. The experiment is described in Section 8.2.1.

7.3.2 Non-uniform distribution choice

Not all non-uniform data is alike, and there are many ways to distribute the frequencies of the characters in the alphabet. The wavelet tree has applications within full text indexing which suggests that using a distribution based on how words are distributed within a normal English text could be a good choice for testing purposes, because it would be a realistic use case. Zipf's Law describes such a distribution [19, abstract] but it requires a distribution parameter s that describes the frequency relation between each symbol, e.g. if the most frequent word has double the frequency of the 2nd most

frequent word and this relation continues down the list of most frequent words, it is a Zipf's Law distribution with an s parameter of 2. We have not been able to find anyone that describes which parameter value would produce a distribution most closely resembling real world English. We have searched through various articles to find an s value representative of the English language, but it does not seem like there is a single good value as it depends a lot on the type of text, e.g. scientific journals vs. newspapers vs. books. This is also a conclusion that Piantadosi [19, abstract] arrives at.

It is possible to estimate s for a given text but doing so then creates the problem of choosing a representative text to estimate s from. We tried using the word frequencies in the NGSL [20] wordlist which contain the 31,241 most used words in the English language and the frequency with which they appear, to estimate s. NGSL is based on data from the Cambridge English Corpus which is a multi-billion word collection of written, spoken and learner texts and is the largest of its kind. This fact combined with the fact that NGSL is fairly new (2013) makes us assume that the frequencies in the NGSL wordlist are accurate enough for our purpose.

Estimating s using a subsequence of words from NGSL gave us a value close to 1 and it only grew closer to 1 the more words we used from the list. This is a problem because with an s parameter of 1, a Zipf's Law distribution is uniform. It also tells us that the English language, or at least the part that has been aggregated in NGSL, might not, in fact, be following the Zipf's Law closely, as we would expect our calculations to converge towards some constant value > 1. Instead we use the data from the NGSL word list and generate our own non-uniform dataset based directly on the word frequencies found in the NGSL word list. This way we end up with a more realistic non-uniform dataset than if we had used the Zipf's Law model since the data is based on real empirical data and we avoid the problem of choosing a good s value.

7.4 Choice of Query Parameters

It is important to ensure that we do not introduce a bias in our experiments on the rank and select query performances by our choice of query parameters. As we have chosen to use a randomly generated input string with uniform distribution of characters for most of our tests, there should be little difference in the frequency of characters and little difference in query performance based on the exact choice of character. There is, however a difference of where in the tree the node each character corresponds to, and we should make sure to use characters from various positions in the alphabet, to have the queries together traverse as much of the tree as possible, to avoid caching hiding the actual performance of the queries.

For the rank queries there is also the position parameter, determining how far into the string the query should look and therefore how far into each bitmap the query should look. A high value (close to the length of the string) might seem like a good idea to make the query go through most of the bitmaps, but we do not want to introduce a bias by using some constant high value, nor do we want to risk introducing a bias by only looking at high values for the position parameter. Again we choose to use values from all parts of the range of valid values for the parameter.

We are also interested in avoiding introducing any bias by using only one type of combination of parameters. If we had e.g. let both parameter values depend on the index of a single for-loop around the call to the query, we would have only tested low character values together with low position values and high character values together with high position values.

Instead we let one parameter ascend from valid low values to valid high values with even spacing to reach the highest valid value in the lastly performed query. Meanwhile, the other parameter increases more rapidly with wider spacing, and then wraps around before passing highest valid value to then start again at low values, with an offset to not repeat parameter values, doing so many times before the end. This ensures the queries are performed for all combinations of high, medium and low parameter values in our experiments.

7.5 Tools Used

We have looked at and tested the capabilities of several profilers and tools for determining the number of cache misses, branch mispredictions and translation lookaside buffer misses.

7.5.1 Tools

We have used these tools to count cache misses, branch mispredictions, etc., measuring memory usage and finding hotspots in our code.

Perf [5] is a performance analysing tools primarily implemented in the Linux kernel, available from version 2.6.31. It supports reading and reporting various counters from the *hardware*, meaning it does not emulate the CPU or anything similar, as some other tools like Callgrind do. It can profile the entire system or a specific process, but not subsections of a program.

PAPI [6], short for Performance Application Programming Interface, will use the perf kernel driver when available but itself pre-dates perf. It requires the analysed program itself to set up and initialize PAPI, but therefore also supports starting and stopping the counter data collection at specific points in the program, enabling profiling of subsections of the program.

Massif [7] is a heap profiler. It can count how much heap memory a program is using during its run by recording calls to `malloc`, `calloc`, `realloc`, `memalign`, `new`, `new[]`, and other similar functions. It then gathers them in a number of snapshots and detailed snapshots, which can be scheduled by the program itself too. Massif can be useful for finding how much memory a program uses and which parts of a program uses most memory.

[5] `perf.wiki.kernel.org/index.php/Main_Page`
[6] `icl.cs.utk.edu/papi/software/`
[7] `valgrind.org/docs/manual/ms-manual.html`

Callgrind [8] is a callgraph analyser tool in the Valgrind suite. It supports wrapping a single program. Valgrind compiles the analysed program into an intermediate representation and runs that completely in a virtual machine to extract information for its tools. This causes the program to run much slower while being analysed, but this is a minor concern for us because it only increases the time it takes to tun our experiments but does not affect the results. Callgrind outputs a .callgrind file which can then be viewed in the kCacheGrind GUI program. We use it for finding hotspots in our code; which parts our program is spending most of its time in and therefore which parts we should try to optimize.

We calculate the CPU clock cycles using the `PAPI_TOT_CYC` papi event which returns the total number of unhalted CPU clock cycles, including when the CPU clock changes to a higher frequency in what Intel calls "Turbo Boost" or more generally "dynamic overclocking". We expect this value to be more accurate than calculating cycles using `PAPI_get_real_cyc()` which estimates the cycles based on wall time[9] and this means that `PAPI_get_real_cyc()` depends on the Time Stamp Counter (TSC) frequency which is constant. Because of this the TSC frequency is not based on CPU frequency in any way and because work gets done at CPU frequency and not TSC frequency, `PAPI_TOT_CYC` seems like the best choice. This is also recommended by Intel[10].

In Table 1 we have listed the various counters and values we have read using PAPI and their description. Not every test uses every one of these.

Because PAPI does not support gathering all combinations of hardware at the same time, we had to gather Translation lookaside buffer misses, level 2 cache misses, and level 3 cache misses in a separate run of the program with the same parameters. We do not expect that this will have any significant influence on the results of our experiments.

[8] `valgrind.org/docs/manual/cl-manual.html`
[9] `icl.cs.utk.edu/projects/papi/wiki/PAPIC:PAPI_get_real_cyc.3`
[10] `software.intel.com/en-us/articles/measuring-the-average-unhalted-frequency`

Table 1: PAPI counters and data sources we used and their description

PAPI Source	**Description**
`PAPI_get_real_cyc()`	Real Cycles / Wall Time Cycles
`PAPI_get_real_usec()`	Wall Time (microseconds)
Event `PAPI_TOT_CYC`	Total Cycles
Event `PAPI_L1_DCM`	Level 1 data cache misses
Event `PAPI_L2_DCM`	Level 2 data cache misses
Event `PAPI_L3_TCM`	Level 3 total cache misses (level 3 data cache misses was unavailable)
Event `PAPI_L2_DCH`	Level 2 data cache hits (hits only available for level 2)
Event `PAPI_BR_MSP`	Conditional branch instructions mispredicted
Event `PAPI_BR_CN`	Conditional branch instructions in total
Event `PAPI_TLB_DM`	Data translation lookaside buffer misses
`PAPI_get_dmem_info()`	Memory information as `meminfo` object
`meminfo.size`	Size of Memory used
`meminfo.resident`	Size of Resident memory used
`meminfo.high_water_mark`	Size of Peak memory usage

Part III
Algorithms & Experiments

This part of the thesis deals with what we have implemented, what optimizations we have made to reduce cache misses, branch mispredictions and translation lookaside buffer misses, and how we have tested and analysed these optimizations and their effect on the resulting running time and memory usage.

8 Simple, Naïve Wavelet Tree: Rank and Select

This section deals with the simple, straightforward, naïve implementation based on the description by Navarro [1, Section 2], before any smart ideas and optimizations were introduced. We will call this version of the wavelet tree SimpleNaive.

The construction of the wavelet tree is implemented similarly to the pseudo-code in Section 3.1. In our implementation, alphabets are stored as two integer values: a minimum and a maximum. It is explained in Section 6.1 how this is equivalent to storing the full alphabet and passing pointers into it around. *Bitmap* is stored as a `vector<bool>` which is a tightly packed data structure, only using 1 bit per bool[11], plus a little bookkeeping data and at most 8 bytes minus 1 bit of superfluous stored data when the amount of bits stored does not align with 8 bytes.

Rank queries is implemented as described in Section 3.3 and binary rank is implemented as a simple linear scan of the bitmap. Select queries is implemented as described in Section 3.4 with binary select implemented as a simple linear scan of the bitmap.

8.1 Optimizations

8.1.1 Binary Rank using Popcount

To improve BinaryRank we will use the intrinsic cpu instruction `popcount`, which will count the number of 1s in the binary representation of the number that is passed to it. Our use of `popcount` to improve binary rank and select queries was inspired by González et al. [21] who used it to improving binary rank and binary select for bit arrays. Unlike González et al., we do not use a popcount function implemented in software, but rather the built-in `popcount` instruction in the CPU instruction set, which we assume to be the fastest way to calculate `popcount` since it is only a single instruction implemented in hardware. The built-in `popcount` cpu instruction takes as argument an `unsigned int` or an `unsigned long`. The `vector<bool>` stores the bits in a backing array of `unsigned long`s and a pointer to the desired position in this array can be retrieved from the `vector<bool>`. The implementation will therefore be working on `unsigned long`s and we will call their size (64 bits on machine 1) our *wordsize*. When using `popcount`, BinaryRank remains in theory an $O(\frac{n}{wordsize}) = O(n)$ operation, as

[11] `http://www.cplusplus.com/reference/vector/vector-bool/`

wordsize is a constant factor, but it has a large practical effect on performance as can be seen in Section 8.2.3.

To use `popcount` we call `__builtin_popcountl` which is a function built into the GCC compiler[12]. It takes an `unsigned long` as a parameter and returns the number of 1s in it. `__builtin_popcountl` will automatically figure out how to do popcount based on what CPU you are using. Popcount as an intrinsic cpu instruction is supported on both AMD[13] and Intel arhitectures[14]. We have verified, by looking at the produced assembly code, that `popcount` is calculated using the cpu instruction `popcnt` on our test machine.

The binary rank can then be found by summing the result of calling `popcount` on each word of the bitmap up to a given position p. When the position argument of the rank query is not a multiple of the word size, it is necessary to constrain what part of the last word is counted using `popcount`. This can be done by constructing a bitmask by bitshifting the number 1 p times towards the most significant bit and then subtracting one, as that will create a word where the p least significant bits are set to 1 and the rest to 0. Then we do a bitwise `AND` operation of this bitmask and the word containing the bit corresponding to p, and call `popcount` on the result. As an example, assume a word contained the bits 10101 but we were only interested in the 3 least significant bits of the word, the 3 rightmost bits in this representation. We could construct the bitmask 00111 by bitshifting 1 three times to the left making it 01000, and then subtracting 1 from it, making it 00111. Doing a bitwise `AND` operation of 10101 and 00111 produces 00101, containing exactly the 3 bits from the original word we were interested in, and replacing the rest with zeroes which will not increment the value of the result of a popcount operation on it. The result is the same as if we could popcount only the three bits we were interested in.

As also noted in [1], we do not need to count the number of 0s, although required by the algorithm, as we can simply take the number of bits in the bitmap and subtract the number of 1s to calculate the number of 0s.

8.1.2 Binary Select using Popcount

We improved Binary Select by again using the `popcount` instruction. We iterate through the words of the bitmap and call `popcount` for each word and sum up the results along the way. When the sum after the next word would be greater than the sought number of occurrences we discard the `popcount` result for the next word and fall back to the simple binary select for that next word to find the position within that word.

If we define the input occurrence parameter as o, the number of words iterated through so far as w, the sum so far as *sum*, and the wordsize as *ws*, then the occurrence argument for that last simple binary select is then the $o - sum$ and the output position is $w \times ws + \text{BinaryRank}(bitmapwords[ws + 1], o - sum)$ [15].

[12] `gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Other-Builtins.html`
[13] `support.amd.com/TechDocs/24594.pdf`
[14] `software.intel.com/sites/landingpage/IntrinsicsGuide`
[15] Using simple binary rank without `popcount`

Again, when popcount for 0s instead of 1s is needed, we simply subtract the result of `popcount` from 1 to obtain the count of 0s.

8.2 Experiments

8.2.1 Uniform vs. Non-Uniform data

We have tested and graphed the build wall time as well as the rank and select query wall times in Figure 11. The non-uniform data has been generated by extracting word frequencies from the NGSL [20] word list, then generating a 10^8 characters long string using an alphabet of integer characters the size of the NGSL word list, each character with the corresponding frequency from the word list, but randomly permuted so each frequency of character occurs at a random place in the alphabet. The frequencies remain the same, only the position of the frequencies in the alphabet changes. The permutation was done to avoid the bias of having all the most frequent characters in the beginning of the alphabet and thus in the leftmost side of the wavelet tree.

In Section 7.3.1 we theorized that building the tree on non-uniform data would be slightly faster as some of the characters would not occur in the string and therefore some of the nodes in the tree would not have to be created. We also theorized that much the same would happen for Select queries as it could terminate much faster when a character could not be found in the tree. We did not expect it would make as much difference for rank queries as they cannot terminate as early as select queries for non-occurring characters.

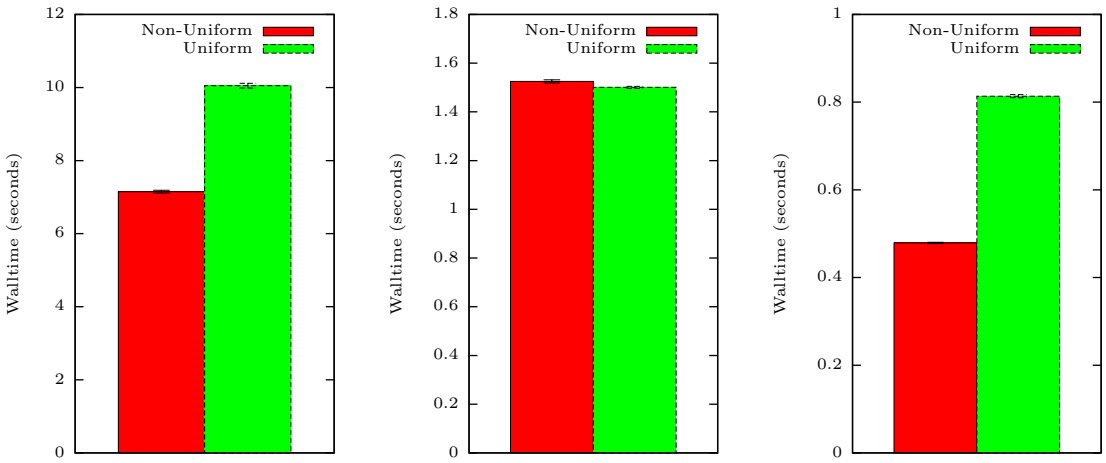
Looking at Figure 11 we see that our theories have been confirmed. The build time in Figure 11a is noticeably lower for the non-uniform data. The select query time in Figure 11c is almost half for non-uniform data as that of uniform data. The rank wall times are much more similar and it is on uniform data that it is slightly faster, but only by about 1.6 %. We expect this is because the non-uniform data just so happens to be distributed so that some of the query parameters used in our test result in a slightly slower execution compared to the uniform data. When looking at the numbers of branch mispredictions and cache misses and so forth, we find that the rank queries on the non-uniform data have about 4 % more level 3 cache misses, and even less difference in the other measurements.

8.2.2 Running time of Tree Construction vs Alphabet Size

We would like to find out whether our implementation of the construction of the tree conforms to the theoretical running time of $O(n \log \sigma)$ and how much of an improvement using the `popcount` cpu instruction was for the queries.

The general test setup is as described in Section 7.2. The query parameters were chosen as described in Section 7.4.

To test what Big-O notation running time our construction algorithm was running at, we tested the running time of building the tree relative to the alphabet size by running the program 5 times for each size of the alphabet, and took the average value of the



(a) Build Wall Time **(b)** Rank Wall Time **(c)** Select Wall Time

Figure 11: Build time and Rank and Select query time for uniform and non-uniform data based on the NGSL word list.

resulting measurements for each measurement type we used. We tested for alphabet sizes 2^p with $p = [8...23]$ and used a constant input string of length $n = 10^8$ characters, except in a single test (Figure 12e) where we used $n = 10^2$.

A theoretical running time of $O(n \log \sigma)$ is equivalent to $a \cdot n \log \sigma$ where a is some constant factor. Assuming our construction algorithm has this running time, a plot of the wall time divided by $n \log \sigma$ should converge on the constant factor a as $\sigma \to \infty$ In Figure 12a we have plotted this, and find that it could be said to be converging on a constant value until reaching an alphabet size of about 2^{16} whereafter it increases as σ increases, with what looks like exponential growth. This means our implementation of the construction of a wavelet tree is not conforming to the theoretical running time for higher alphabet sizes.

To attempt to understand why our algorithm performs so, we turn to the many other measurements available to us through PAPI: branch mispredictions, cache misses, etc.

Looking at the raw wall time and branch misprediction numbers in Figure 12b it might seem natural to conclude that the branch mispredictions are to blame.

But if we instead plot the rate of branch mispredictions, as we have done in Figure 12c, we can see that the rate of branch mispredictions stay constant for most of the tested alphabet sizes, and even decrease for large alphabet sizes.

We next turn to look at cache misses, plotting all three levels in Figure 12d and see that cache misses increase for larger alphabet sizes up to an alphabet size of around 2^{18} after which they seem to remain constant for even larger alphabet sizes. This is in contrast to the wall time over theoretical running time plot in Figure 12a that seem to remain somewhat constant until about 2^{18} after which it increases. We can conclude that the cache misses are not the problem.

We then considered that the difference in theoretical and practical running time

might be our algorithm spending a constant amount of time per node, constructing it. This factor would be independent of the size of the input, n, and scaling linearly with alphabet size, σ as that determines the number of nodes in the tree. If so, the actual running time should then be $a \cdot n \log \sigma + b\sigma$. Since n in our previous experiment is somewhat large (10^8), it might be the dominating factor in the running time. So to show whether the added $b\sigma$ term can explain the running time, we redid the experiment with a reduced length input string $n = 10^2$ and plotted it in Figure 12e divided by $\log(\sigma) + \sigma$ to see whether it would converge on some constant as $\sigma \to \infty$.

We can see in Figure 12e that it does not converge on any constant value other than 0, meaning the constant factor from each node cannot explain our implementation's running time.

Having not found an explanation we pull data for translation lookaside buffer misses from the experiment and plot it together with wall time, both divided by $\log \sigma$, in Figure 12f

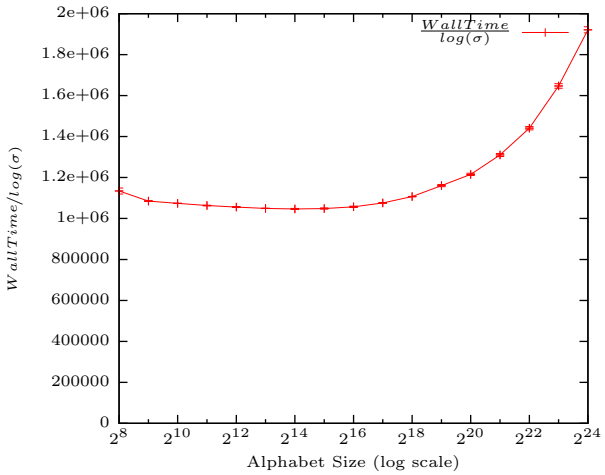
We can see that the TLB Misses increase drastically from alphabet size about 2^{20} and up. Having found no other reasonable explanation for the discrepancy between the theoretical and our implementation's actual running time, we find it probable that the TLB Misses are the culprits here.

In our further experimentation of the further optimization attempts we do, we will be using an alphabet size of 2^{16}. It is a realistic use case to use a type such as `char`, `wchar16_t` or `wchar32_t` which are stored in 8, 16 and 32 bits respectively. `char`'s size of 8 bits corresponds only to the ASCII table with 256 entries and we believe that many real-world scenarios require a larger alphabet. `wchar16_t` enables an alphabet up to $2^{16} = 65,536$, which should be enough for many use cases, such as full text indexing. Zachery B. Simpson[16] has found that no book occurring in the Gutenberg Project uses more than 43113 distinct words. According to one website[17], testing suggests an average adult has a vocabulary of 20,000 - 35,000 words. Others[18] [19] cite researchers saying about 60,000 words is the actual limit when including names. Whichever is the actual number, they all suggest that an alphabet size of 2^{16} is sufficient to index all the occurring words in a realistic use case text.

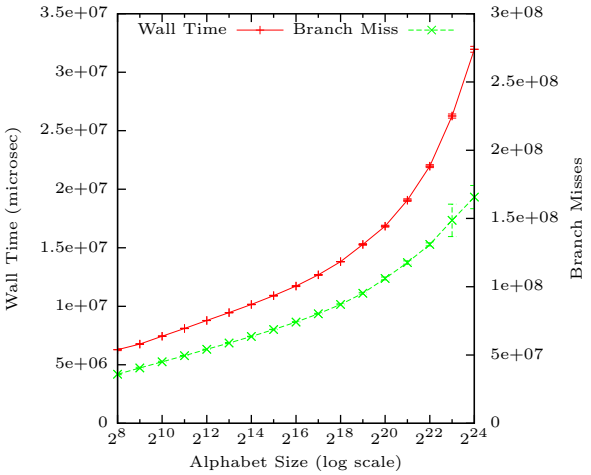
Looking at the graphs from this experiment we can see that building the tree using an alphabet size of 2^{16} is still fairly quick, not running into much trouble with TLB misses, and not exceeding the expected asymptotically bound running time.

We also attempted using an alphabet size of 32, but our machine did not have enough memory for that to be possible on a sizeable input string.

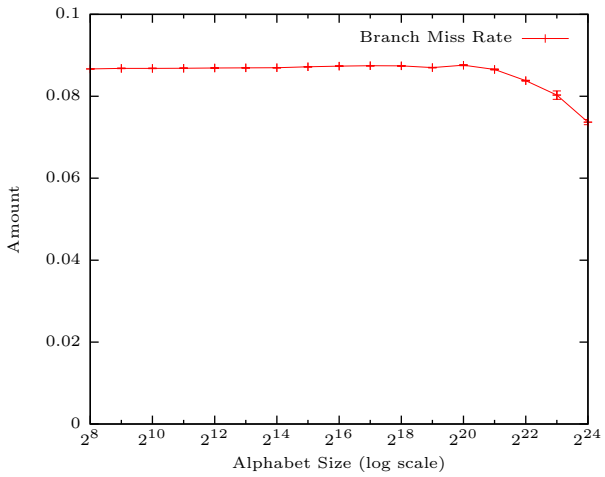
[16] `mine-control.com/zack/guttenberg/`
[17] `testyourvocab.com/blog/2013-05-10-Summary-of-results`
[18] `worldwidewords.org/articles/howmany.htm`
[19] `english.stackexchange.com/questions/93289`



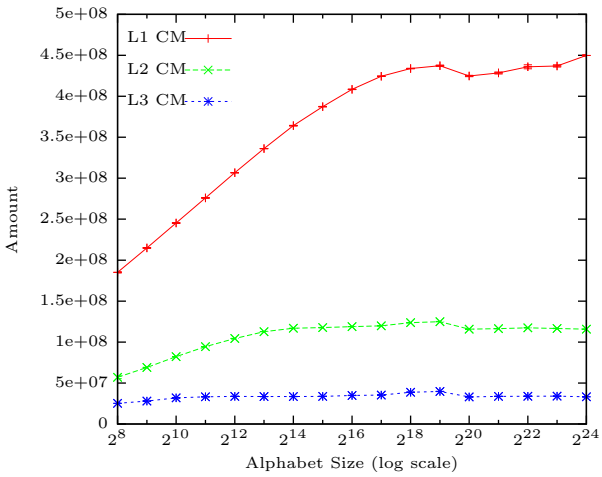
(a) Wall Time divided by theoretical running time



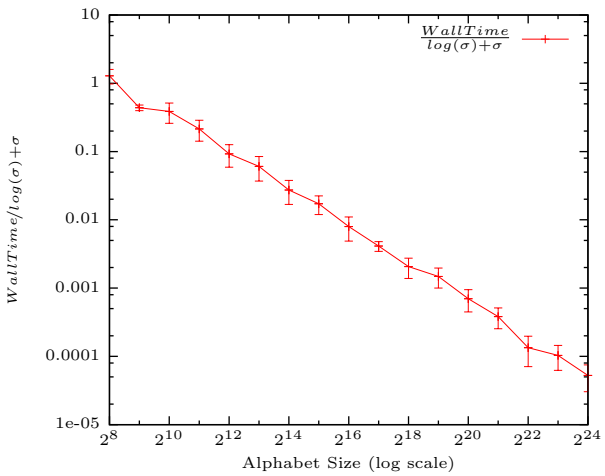
(b) Wall Time and Branch Mispredictions



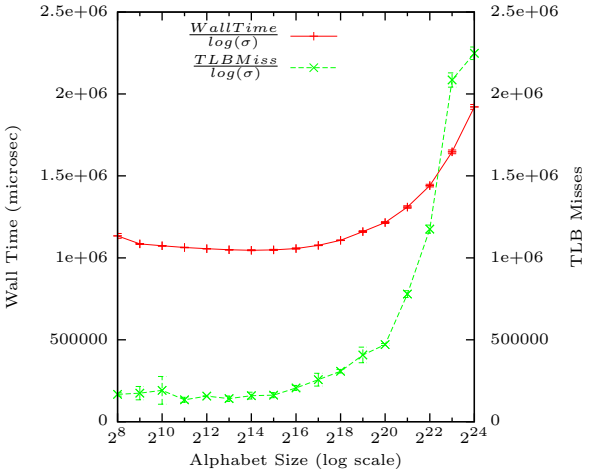
(c) Branch Misprediction Rate



(d) Level 1-3 Cache Misses



(e) Wall Time divided by $\log(\sigma) + \sigma$



(f) Wall Time and Translation Lookaside Buffer Misses divided by theoretical running time

Figure 12: Various Measurements for the construction of the SimpleNaive Wavelet Tree over various alphabet sizes.

8.2.3 Rank and Select using Popcount

We wanted to see how much of an improvement using the native cpu instruction `popcount` was, and how it affected the cache misses, branch mispredictions and TLB misses.

In Figure 13a, Figure 13b, and Figure 14 we see the resulting relative cpu cycles, wall time, branch mispredictions, translation lookaside buffer misses, and cache misses for our rank and select queries, respectively. We have chosen the measured values of the queries not using popcount to be index 100 and calculated and plotted the relative values of the queries using popcount, to show which values increase or decrease by relative amounts, all within the same graph. In Figure 14 we list the actual raw values as well as the percentages graphed in Figure 13a and Figure 13b.

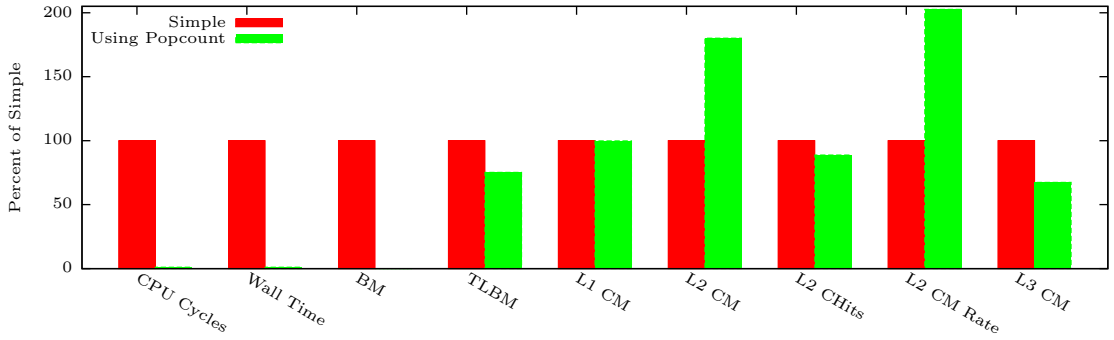
In all three figures we see that the algorithm using `popcount` is much faster, using only a fraction of the time of the other algorithm, about 0.011 % for rank and 0.0051 % for select. We see a massive decrease in branch mispredictions for both rank and select queries. For the select queries we see a great reduction in translation lookaside buffer misses as well as cache misses, especially Level 2 and 3. For the rank queries, we see some improvement in TLBM and L1 CM and a slightly larger improvement in L3 CM, but we also see a high increase in L2 CM rate to more than double. The higher L2 cache miss rate comes from both having fewer L2 hache hits and many more L2 cache misses. We have no good explanation to offer as to the L2 cache miss rate increases so much, other than the algorithm being different and having a different access pattern. It is not a problem at any rate, as we can see in the massive decrease in running time.

We believe the massive reduction in branch mispredictions accounts for some of the saved cpu cycles. Agner Fog has tested the Ivy Bridge architecture and found that the branch misprediction penalty is "15 cycles or more"[20]. We have not tested this claim ourselves, but choose to trust him, as we only use it to get an approximate percent value of how much of the difference in running time the branch mispredictions can account for. Given that the branch misprediction penalty on the Ivy Bridge architecture, on which this experiment was run, is about "15 cycles or more"[20], we can calculate an estimate of how many cpu cycles the branch misprediction reduction has saved us. The number of saved branch mispredictions for RANK is then $1.40 \cdot 10^9 - 2.27 \cdot 10^4 = 1.40 \cdot 10^9$ mispredictions. Assuming a penalty of 15 cycles this becomes $1.40 \cdot 10^9 \times 15 = 2.10 \cdot 10^{10}$ cpu cycles saved, and given that the total number of cycles saved is $3.97 \cdot 10^{11} - 4.42 \cdot 10^9 = 3.93 \cdot 10^{11}$, it is $2.10 \cdot 10^{10} / 3.93 \cdot 10^{11} = 0.0533 = 5.33\,\%$ of the total amount of cpu cycles saved. This means that the branch mispredictions do have an effect, but it is only a small part of this increase in speed. The main improvement, we expect, comes from using only a few cpu cycles per word of the bitmap to calculate the binary rank, as well as possibly the slight decrease in L1 and L3 cache misses.

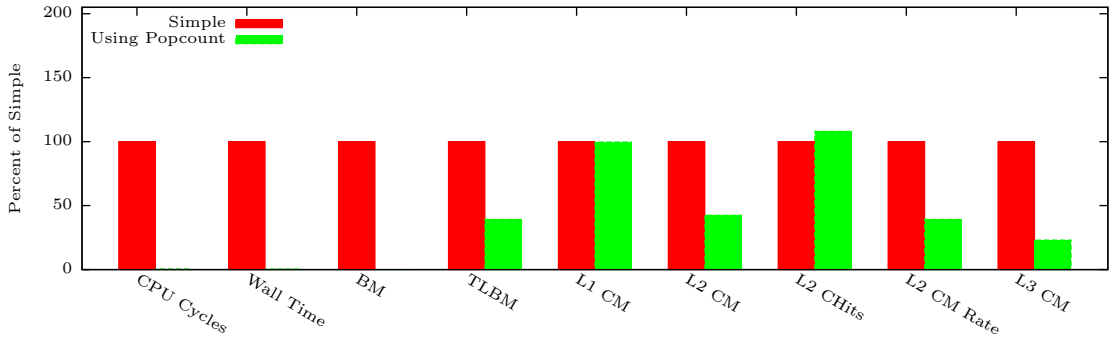
By similar calculations the saved cpu cycles from branch mispredictions for SELECT is at least 48,84 % of the total saved. We expect this is because of the much higher number of branch mispredictions and the lower number of cycles for the original select algorithm.

[20] Section 3.7 in `http://www.agner.org/optimize/microarchitecture.pdf`

Looking at the values in Figure 14 we find that, of the measurements we collect, cache misses are among the highest and were not reduced significantly by using the `popcount` instruction. Cache misses are expensive and reducing them could greatly increase the speed of queries on the wavelet tree.



(a) Rank



(b) Select

Figure 13: Rank and select queries using simple binary rank and select vs. rank and select queries using binary rank and select using the popcount instruction. Y-Axis is index 100 of the simple queries, that is, every value is percent of the value for the simple query.

9 Precomputing Binary Rank in Blocks

Using the profiling tool callgrind, we concluded that most of the work during queries is performed inside each node, calculating the binary rank of each bitmap. Rank queries are simply summing up results of popcounting each word, and we considered whether precomputing these sums for blocks spanning several words, covering the bitmaps could improve the query times. When a block does not line up with the position of a rank query is for, the algorithm can simply fall back to doing popcounting of either the remaining uncovered words or the extraneously covered words, whichever has fewer words. See Section 9.1.1 for more explanation of this.

The rank values can be precomputed easily and cheaply by doing so as the tree is

Figure 14: Values for Figure 13a and Figure 13b

Rank	no `popcount`	`popcount`	Percent
CPU Cycles	3.97e+11	4.42e+09	1.113 %
Wall Time	1.33e+08	1.49e+06	1.115 %
BM	1.40e+09	2.27e+04	0.002 %
TLBM	5.65e+05	4.25e+05	75.306 %
L1 CM	1.76e+08	1.76e+08	99.935 %
L2 CM	1.68e+07	3.03e+07	180.189 %
L2 CHits	1.59e+08	1.42e+08	88.883 %
L2 CM Rate	0.11	0.21	202.726 %
L3 CM	1.59e+07	1.08e+07	67.497 %

Select	no `popcount`	`popcount`	Percent
CPU Cycles	1.01e+12	5.22e+09	0.517 %
Wall Time	3.39e+08	1.76e+06	0.518 %
BM	3.27e+10	3.06e+05	0.001 %
TLBM	1.34e+06	5.27e+05	39.326 %
L1 CM	1.28e+08	1.28e+08	99.782 %
L2 CM	1.68e+07	7.16e+06	42.530 %
L2 CM	1.12e+08	1.21e+08	108.114 %
L2 CM Rate	0.15	0.06	39.338 %
L3 CM	1.57e+07	3.62e+06	23.110 %

built where each individual bit of the bitmap already needs to be computed and stored. The algorithm increments a counter for the corresponding block each time it sets a bit to 1 in the bitmap.

The size of the precomputed blocks, b, is a new variable that could have influence on the running time and memory usage. Some advantages of larger blocks is less memory usage and fewer precomputed value lookups for the same part of the bitmap. Some advantages of smaller blocks are that they can cover more precisely the part of the bitmap that is relevant to the query, leading to fewer calls to popcount. Later, in Section 9.4, we will analyse how the optimal block size b depends on the input size n, and later again we will experiment with varying block sizes to see how it works in practise for a wavelet tree and to see how it corresponds to the theoretical analysis.

To further reduce the space used by the precomputed values, we considered concatenating all the bitmaps into one big bitmap and keeping a single vector of precomputed rank values for blocks for the entire bitmap. That would eliminate the many cases where the length of a bitmap does not align with the block size and the last precomputed value for that bitmap will therefore not cover an entire block, leading to more precomputed values than minimally needed to cover all the bitmaps.

We also considered the cost of TLB misses and how ensuring that entire pages are skipped as often as possible might increase the query performance. We try to achieve this by page-aligning the blocks.

We will test and compare the Rank and Select running times and memory usage of four wavelet trees using precomputed rank values for blocks: one using concatenated

bitmaps and aligned blocks, one using concatenated bitmaps but not aligned blocks, one only using aligned blocks and one using neither.

9.1 Concatenating the Bitmaps

The bitmaps are allocated as one giant bitmap the size of the maximum possible size required to store all the bitmaps for all the nodes. They are stored in a Depth-First-Search-right (DFSr) manner, that is, the bitmap of the right child of a node comes right after the bitmap of the node.

There are many other alternative memory layouts that might provide better performance. The one we expect to have the greatest potential for improving the wavelet tree performance, is the van Emde Boas memory layout [14, Abstract] because of its cache-oblivious nature. It would be a challenge to implement, because the size of a bitmap of a node is unknown before the bitmap of the parent has been calculated, and so the position of each bitmap in the giant bitmap cannot be known before the parent of each has been calculated, meaning construct the tree must be constructed in vEB layout order. To support construction of the wavelet tree in vEB layout order our construction algorithm would have to be dramatically changed, possibly utilizing concurrency constructs such as a job queue and message passing, and so we skip this work for now and save it for future work in Section 13.2.

The sum of the size of all bitmaps on one layer of the tree can at most be n and we can at most have $\log \sigma$ layers, so the maximum size becomes

$$n \log \sigma \ ,$$

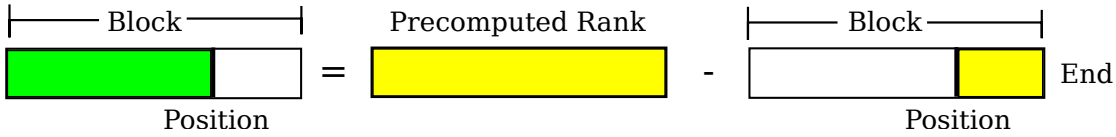
where n is the number of characters in the string and σ is the alphabet size. Luckily for us, memory allocation in Linux does not actually take up space because Linux uses optimistic memory allocation which means that only when the memory is accessed will the actual physical memory be used [21]. This fact enables overcomitting which allows allocation of more memory than what is available which can be a problem for long running processes. As a result we can conclude that allocated memory is not present in physical memory before it is actually initialized. The effect of an optimistic memory allocation scheme and overcomitting is tested by Andries Brouwer [22] who confirms that it is possible to allocated more memory using `malloc()` than what is physically available. So over-allocating the bitmap should not take up any more space than what will actually be needed.

An offset and a size for the bitmap is the stored in each node, so it is possible to index into the giant bitmap and access the bits corresponding to the node. This should also cause a decrease in memory usage as the offset and size are stored in an unsigned long and unsigned int respectively, taking a total of $64 + 32 = 96$ bits per node, where each individual bitmap requires storage of at least a pointer to it, a point to where its internal array starts and a pointer to where it ends, taking up $3 \times 64 = 192$ bits per

[21] `http://man7.org/linux/man-pages/man3/malloc.3.html`

[22] `http://www.win.tue.nl/~aeb/linux/lk/lk-9.html`

Figure 15: Rank value of a part of a bitmap is equal to the precomputed value for the block minus the rank of the other remaining part.



node. Additionally, when using an individual bitmap for each node, they would have been word-aligned, and the bits between the last used bit and the end of the last used word would have gone unused and so, wasted.

A vector is also allocated to hold the precomputed block values of size

$$VectorSize = \frac{BitmapSize}{b}.$$

Instead of storing a pointer to the bitmap and precomputed values vector in each node, they are stored once for the whole tree and then passed down through the query methods when the tree is queried.

In order to be able to index into said vector, integer division of the bitmap offset and the block size is used. It is an efficient and simple way to precompute the rank values of blocks of fixed size of the bitmap, as we do not have to traverse the tree again.

9.1.1 Edge Cases

The rank of a string can be expressed as the sum of the rank of any number and various sizes of subparts as long as they together perfectly cover the string and do not overlap. Because the blocks must perfectly cover the string and not overlap, and the bitmaps of each node are not of same size, nor multiples of some single value, we have a problem if we want to use uniformly sized and distributed blocks. The problem exists at the boundary between bitmaps, where the precomputed rank value will be the sum of the rank of the end of the first bitmaps and the rank of the beginning of the second bitmap.

Looking at a single bitmap for a node, there is an edge case for the first and last part of the bitmap, because they do not fill an entire block, so the corresponding precomputed value cannot simply be used as-is. Instead, the rank of the part of the block that the bitmap does not fill can be computed and then subtracted from the precomputed rank value. This is only worth doing when the bitmap fills more than half a block, because then the other part is smaller than half a block and therefore quicker to compute. Figure 15 illustrates this.

9.1.2 Page-aligning the Blocks

Translation Lookaside Buffer misses are expensive and to avoid those, we can try to reduce the number of pages that is loaded. Using concatenated bitmaps and the precomputed vector of ranks, we only need to load pages of the bitmap at the beginning and

end of each node's bitmap, to compute the popcounted version of binary rank directly on the bitmap, and only within one block at each end. If the blocks are not aligned with the memory pages, then, even if the block size is less than a page, it might span more than one page and thus more than one page of memory must be loaded into the TLB. More precisely, the algorithm might at most load

$$2 \cdot \left(\left\lceil \frac{b}{pageSize} \right\rceil + 1 \right)$$

pages to do the popcount binary rank computation at the beginning and end of each node.

If the blocks are page-aligned, and has block sizes divisible by the page size or that the page is divisible by, the extra +1 will disappear, because a block can no longer span more pages than the number of pages its size is a multiple of. This means we can ensure that at most

$$2 \cdot \left\lceil \frac{b}{pageSize} \right\rceil$$

memory pages of the bitmap are loaded for each node by page-aligning the blocks. With an alphabet size of 2^{16} this amounts to saving up to 16 page loads per query.

While we expect that this will save some expensive TLB misses it also has some drawbacks, especially when not using concatenated bitmaps. For a wavelet tree not using concatenated bitmaps, using page-aligned blocks will cause the first precomputed value of each non-page-aligned bitmap to not cover an entire block, increasing the number of precomputed values needed to cover the bitmap as well as additional computations to calculate exactly which part of the bitmap it covers. For the wavelet trees using concatenated bitmaps, these computations are needed regardless of block alignment, as the blocks are already not aligned with the individual bitmaps, and so we expect that it is an improvement to page-align the blocks in this case. We test this by implementing a variation of the wavelet tree using concatenated bitmaps that does not page-align the blocks.

We test whether it increases the performance of rank and select queries when not using concatenated bitmaps in Section 9.5.1.

9.2 Select Queries with Precomputed Ranks

Select queries, although they do not return a rank value, can still utilize the precomputed rank values to skip much computation directly on the bitmap by iterating though them. Partway in a select query, if the sum of the occurrences found so far and the rank value of the current block of the bitmap is more than the queried-for occurrence, we can add that rank value to our occurrences seen so far and skip ahead to the next block and perform the same test. If the sum is less than the queried-for occurrence, we know the occurrence will be found in the current block and the previously implemented method of calculating select using popcount can then be used, starting at this block.

9.2.1 Edge Cases

As with rank queries, there are edge cases at the beginning and end of each bitmap. However, in this case, the edge case at the end is easily handled as the test of sum of rank and occurrence-so-far should fail, sending the algorithm into the block with the previous select query method, finding the occurrence with no problem and no specific handling of the edge case. This is assuming the input occurrence parameter is valid, meaning that at least that many occurrences of that character is in the original input string.

For the other case, at the beginning of the bitmap, almost exactly the same as in the case for the rank query is done. In fact, a rank query is used to calculate the rank of the first part of the bitmap, using the trick of subtracting from the precomputed value if larger than half a block size, to figure out whether the occurrence is in the first part of the bitmap, and therefore whether a select query should be run on it or not.

Using Rank Queries in Select Queries

We would like to analyze whether it is worth using a rank query to find out whether we should do a select query on the first part of the bitmap when using concatenated bitmaps, as the rank query is purely extra work in the cases where the occurrence is in that part of the bitmap.

We will assume an equal number of occurrences in the string of each character in the alphabet, a uniform distribution of each character in the string, and an equal probability of each valid parameter for the select query. A valid character parameter is one that exists in the input string and a valid occurrence parameter is an integer above 0 and below or equal to the number of occurrences appearing in the input string.

The rank query is a computation of worst-case cost $O(\frac{b}{2})$ because it will at most popcount half of the block, because the precomputed rank value is utilized when advantageous. The select query using popcount in the first partial block is an operation of worst-case cost $O(b)$ because it can at most popcount the entire block. So, in the worst case, when the sought-after occurrence is in the first partial block of the bitmap, $O(\frac{b}{2})$ work is wasted, yet when the occurrence is elsewhere, $O(b)$ work is saved. This means that the boundary between where using the rank query becomes a gain or a loss in performance is where the ratio between the number of times the occurrence is to be found in and outside the first partial block of the bitmap is $\frac{2}{3}$. That is, when considering the worst-case query time for both queries, if the occurrence is to be found outside the first partial block of the bitmap more than one third of the time, using the rank query first to see if that partial block should be select queried is a gain in performance.

This is giving the select query a disadvantage in the analysis, even when the partial block is close to a block in size and it might terminate early if the sought-after occurrence of the character if found early in the partial block.

We expect it is an improvement, though it is not 100 % certain, but we will use it going forward for the algorithms using concatenated bitmaps. We have considered doing tests to determine whether it is an improvement for the running time or not, but as we

are under time constraints and we feel we have other, far more interesting, things to implement and test we will not be testing this.

9.3 Extra Space Used by Precomputed Values

Storing the precomputed values requires more memory: one number per block. There are $O(\frac{n}{b})$ blocks per level of the tree, and so an extra memory consumption of $O(\frac{n}{b} \log \sigma)$ words making the total memory consumption $O(n \log \sigma + (\sigma + \frac{n}{b} \log \sigma) \cdot ws)$ bits.

Since each precomputed value cannot exceed the block size in bits, assuming we do not use block sizes exceeding $2^{16} = 65536$ bits, or $\frac{2^{16}}{8} = 8192$ bytes, we can store them in 16-bit unsigned integers, called `unsigned short int` in C++ on our machine. Since the page size on our machine are 4096 bytes we should not use a block size larger than $\frac{8192}{4096} = 2$ pages if we want to use 16-bit unsigned integers.

In Section 9.5.1 we find that the optimal block size is 16384 bits = 2048 bytes = $\frac{1}{2}$ page.

Assuming the precomputed values are then stored as 16-bit unsigned short integers it will only consume an extra 16 bits or 2 bytes per block and there are $\frac{BitmapSize}{b}$ of these blocks when the bitmaps are concatenated. This means, assuming a block size of 2048 bytes, a relative extra space consumption of

$$\frac{2 \cdot \frac{BitmapSize}{b}}{BitmapSize} = \frac{2}{b} = \frac{2}{2048} = 0.0009765625 = 0.098\,\%$$

of the bitmaps, which is even less when considering the total space used including the nodes.

When the bitmaps are not concatenated there is a higher space consumption by the precomputed values, as each precomputed value do not cover an entire block and more precomputed values is therefore needed to cover all the bitmaps. When the blocks are not page-aligned, each node has potentially one precomputed value not covering an entire block at the end of its bitmap. When blocks are page-aligned, there is another precomputed value potentially not covering an entire block at the beginning of each bitmap. The extra space consumption by the precomputed values when not concatenating the bitmaps is therefore limited proportionally to the number of nodes, which is at most $2\sigma - 1$, making it limited proportionally by the alphabet size.

We expect to see a difference in memory usage between using concatenated bitmaps and non-concatenated bitmaps as well as between using page-aligned and non-page-aligned blocks. However, we expect most of the difference to come from the space used by the bitmaps themselves, and therefore a noticeable difference between the data structure concatenating the bitmaps and the others, with a little difference between using page-aligned and non-page-aligned blocks, with the one using page-aligned blocks using most memory.

9.4 Dependence of Optimal Block Size on Input Size

Whether or not using precomputed values in blocks is an improvement in running time of rank queries or not, depends on which block size is used. If the block size is only 1 bit, then there is nothing to be gained from looking up the value via the precomputed rank instead of looking at the bit in the bitmap. If the block size is the same size as the entire bitmap, then it can only be useful when the positional parameter p for the rank query is above half the size of the bitmap, as the rank of a smaller part of the bitmap beyond the halfway point can then be calculated and subtracted.

The work needed to compute the binary rank of a bitmap of size n without using a precomputed value, but using popcount on pieces (machine words) of the bitmap of size ws is $O(\frac{n}{ws})$ to scan the bitmap using popcount up to the word spanning position p and $O(1)$ to calculate the rank up to position p within that word using popcount, making it in total $O(\frac{n}{ws})$.

When using lookups of precomputed values, the analysis is similar. It costs $O(\frac{n}{b}+b)$ to calculate the binary rank when using precomputed values, as it costs $O(\frac{n}{b})$ to scan the blocks, and $O(b)$ to calculate the rank within a single block using popcount. The optimal block size should be one that minimizes this. The derivative of $\frac{n}{b}+b$ is $1-\frac{n}{b^2}$ and its root is $n=b^2$ making the optimal block size $b=\sqrt{n}$. This is only the optimal block size for a single bitmap, and a wavelet tree has many bitmaps of varying sizes n that are lower near the leaves. This means that the best block size in a wavelet tree is either one that varies for each bitmap or, if using a fixed block size, some value below the theoretically optimal block size for the root bitmap.

Later, in Section 9.5.4, we show using a fixed block size b for all bitmaps in a wavelet tree, whether the optimal b does indeed depend on n and whether the practically optimal value of b is below the theoretically optimal size of b for the root bitmap.

9.5 Experiments

We will test and compare the Rank and Select running times of three wavelet trees using precomputed rank values for varying block sizes: one using concatenated bitmaps and aligned blocks named Preallocated, one using concatenated bitmaps but not aligned blocks named UnalignedPreallocated, one using aligned blocks but not concatenated bitmaps, called Naive, and one using unaligned blocks and non-concatenated bitmaps called UnalignedNaive. In table-form:

Name	Concatenated Bitmaps	Page-aligned Blocks
Preallocated	yes	yes
UnalignedPreallocated	yes	no
Naive	no	yes
UnalignedNaive	no	no

Later, we will compare the memory usage and query times with the non-precomputed version called SimpleNaive.

Test Setup
The general setup is as described in Section 7.2. The query parameters were chosen as described in Section 7.4.

9.5.1 Query Running Time for Bitmap with Precomputed Blocks for different Block Sizes

Rank Queries

In Figure 31a we have plotted the rank query wall time in μs for the wavelet trees using precomputed rank values. See Figure 31a in Appendix A for a graph of the same, covering a wider range of block sizes, from 2^6 to 2^{20} bits, showing that the wall time is worse for both smaller and larger block sizes than the ones in Figure 31a.

We see that both concatenating the bitmaps and page-aligning the blocks is consistently slower, which was expected for concatenating the bitmaps, but not entirely so for page-aligning the blocks. Preallocated is on average about 6.22 % slower than Unaligned-Preallocated, so page-aligning the blocks when using concatenated bitmaps is a 6.22 % performance hit. Preallocated is on average about 13.85 % slower than Naive, meaning concatenating the bitmaps when using page-aligned blocks is a 13.85 % performance hit.

Naive has its fastest running time at 1 page per block, whereas both trees using concatenated bitmaps (Preallocated and UnalignedPreallocated) seem to have a slightly better performance with a slightly higher or slightly lower block size, at 0.75 and 1.25 page per block.

UnalignedNaive is the surprising outlier in this graph with a much lower wall time, especially for smaller block sizes. At block size = 0.5 page size, where UnalignedNaive is fastest, it only takes 65.58 % of the time that Naive does at block size = 1 page size, where Naive is fastest.

Much of the increased running time of Rank queries on the two Preallocated wavelet trees can be explained by the increased amount of instructions needed to calculate the rank of the first part inside the first block of each bitmap because the precomputed value includes part of the preceding bitmap, as well as the ineffectiveness of utilizing the precomputed rank values for small bitmaps, for the same reason. This is also corroborated by Figure 16c that plots the number of branches executed. It introduces more branches to the code to check for alignment and to find which part of the giant bitmap corresponds to the current node. We can see that it is the Preallocated tree using both concatenated bitmaps and page-aligned blocks that execute the most branches, while UnalignedNaive executes much, much fewer than all the others. This looks similar to the wall time plot in Figure 31a.

By examining Figure 16b and Figure 16d we can conclude that part of the wall time difference between using and not using concatenated bitmaps is due to the increased number of branch misses from a higher branch miss rate.

In Figure 16d we initially see a surprising increase in the branch misprediction rate of UnalignedNaive for smaller block sizes. When looking at Figure 16b we see that it follows somewhat the same increase in branch misprediction amount for smaller block sizes as the

others, making us conclude that UnalignedNaive has a higher branch misprediction rate alone because it has fewer easily predicted branches but the same amount of branches difficult to predict compared to the others for smaller block sizes. We expect this is a result of UnalignedNaive where it is not necessary to do any calculations to figure out which part and how much of the bitmap the first precomputed value covers, as it always covers an entire block of the bitmap and is perfectly aligned with the start of the bitmap instead of a memory page. This means that a number of if-statements comparing the size of the bitmap with the block size to figure out whether the precomputed value can be used is not present in UnalignedNaive, where they are present in the others.

Looking at Figure 17a we can see that our expressed goal of reducing TLB misses when using page-aligned blocks is achieved when not using concatenated bitmaps, though only little but TLBs are not reduced when concatenated bitmaps are used which is in line with our expectation. On the other hand, we see that TLB misses are reduced to about 29.80 % when using concatenated bitmaps in the page-aligned version and to about 26.39 % when not page-aligning the blocks. This improvement was not enough to make up for the extra bookkeeping code, however, it seems. We also notice that the two trees using non-concatenated bitmaps, Naive and UnalignedNaive has a noticeable drop in TLB misses at a block size of 1 page size.

Looking at the level 1 data cache misses we can see that there are fewest cache misses when a block size is equal to half a page size, with a sharp rise in cache misses again for smaller block sizes. We expect this is the main reason that UnalignedNaive exhibits the best running time at that block size instead of at lower values. The others also seems to have the best level 1 data cache performance for rank queries at half a page per block, while their wall times are best at a full page per block. This might be explained by total branch execution, as we saw in Figure 16c, they execute many more branches at lower block sizes, which we expect to be because of the extra bookkeeping code needed.

The level 2 data cache miss rate plotted in Figure 17e is generally worst for block size = 0.5 page size, but looking at the raw cache misses in Figure 17c we generally see more cache misses for higher block sizes, meaning that the level 2 data cache miss rate cannot explain the better performance at block size = 1 page size for every tree other than UnalignedNaive. Looking at Level 2 data cache misses in Figure 17c and the level 2 data cache miss rate in Figure 17e, we do not see much difference between the different tree implementations, except that the Naive tree has a noticeable drop in both the raw amount and the rate at block size = 1 page size, just like it had for TLB misses. It is interesting, though, that they all have a somewhat high, $0.35 - 0.4$, level 2 data cache miss rate around 0.5-0.75 page per block, where they all have good wall time performance. This could be explained by the good level 1 data cache performance at those block sizes. To explain what we mean, let us assume there is a fixed amount of operations accessing memory that is hard for the cache to have prefetched or otherwise loaded beforehand, independent of the block size. E.g. when the queried-for position is reached in a bitmap and the rank algorithm jumps to a different node and a different bitmap. When the first cache level can handle more and more of the 'easy' memory operations, fewer of those are left to be handled by the second cache level, yet the same

amount of 'hard' memory accesses are hitting the second cache level, and so the miss rate of the second cache level will increase for lower block sizes, but not because of more misses, but because of fewer hits, as can be seen in Figure 17d. In fact, the level 1 cache misses and level 2 cache hits, in Figure 17d and Figure 17b respectively, look near identical.

In Figure 19f we have plotted the level 3 total cache misses. We notice that all four trees have fairly low level 3 cache misses at the lowest tested block size of 0.25 page size. The trees using concatenated bitmaps then rise to have the most level 3 cache misses at around 1 page per block then decreasing again at higher block sizes. What is perhaps most interesting in this graph is that the Naive tree again has a large dip at 1 page per block. We expect this dip, combined with the others, is what causes Naive to have its fastest rank query wall time at 1 page per block.

Select Queries

In Figure 31b we see the wall time of 1000 Select queries for the different wavelet trees using precomputed rank values. Again, see Figure 31b in Appendix A for a graph of the same, covering a wider range of block sizes, from 2^6 to 2^{20} bits, showing that the wall time is worse for both smaller and larger block sizes than the ones in Figure 31b. We can see that all have the best running time at half a page per block, though some of them have about the same speed at 0.75 page per block. We expect the main reason for this is to be found in the level 1 data cache performance data, which is shown in Figure 19b as we can see that they have better level 1 data cache performance at lower values, except for Naive, which has its best performance at 0.5 page per blockand for decreasing block sizes the cache misses increase again.

In much the same way as for rank queries, the branch mispredictions, as plotted in Figure 18b, decrease as block size increases, but looking at Figure 18d we can see that the branch misprediction rate is highest at about 0.75 page per block and much smaller at smaller block sizes, this is because much more branching code, correctly predicted, is executed at block sizes below 0.75 pages per block, as can be seen in Figure 18c.

The amount of TLB misses across block sizes seen in Figure 19a is similar to the TLB misses for rank queries in Figure 17a as we again see that the biggest reduction in TLB misses comes from using concatenated bitmaps and that page-aligning the blocks does make a difference when using non-concatenated bitmaps but no difference when using concatenated bitmaps.

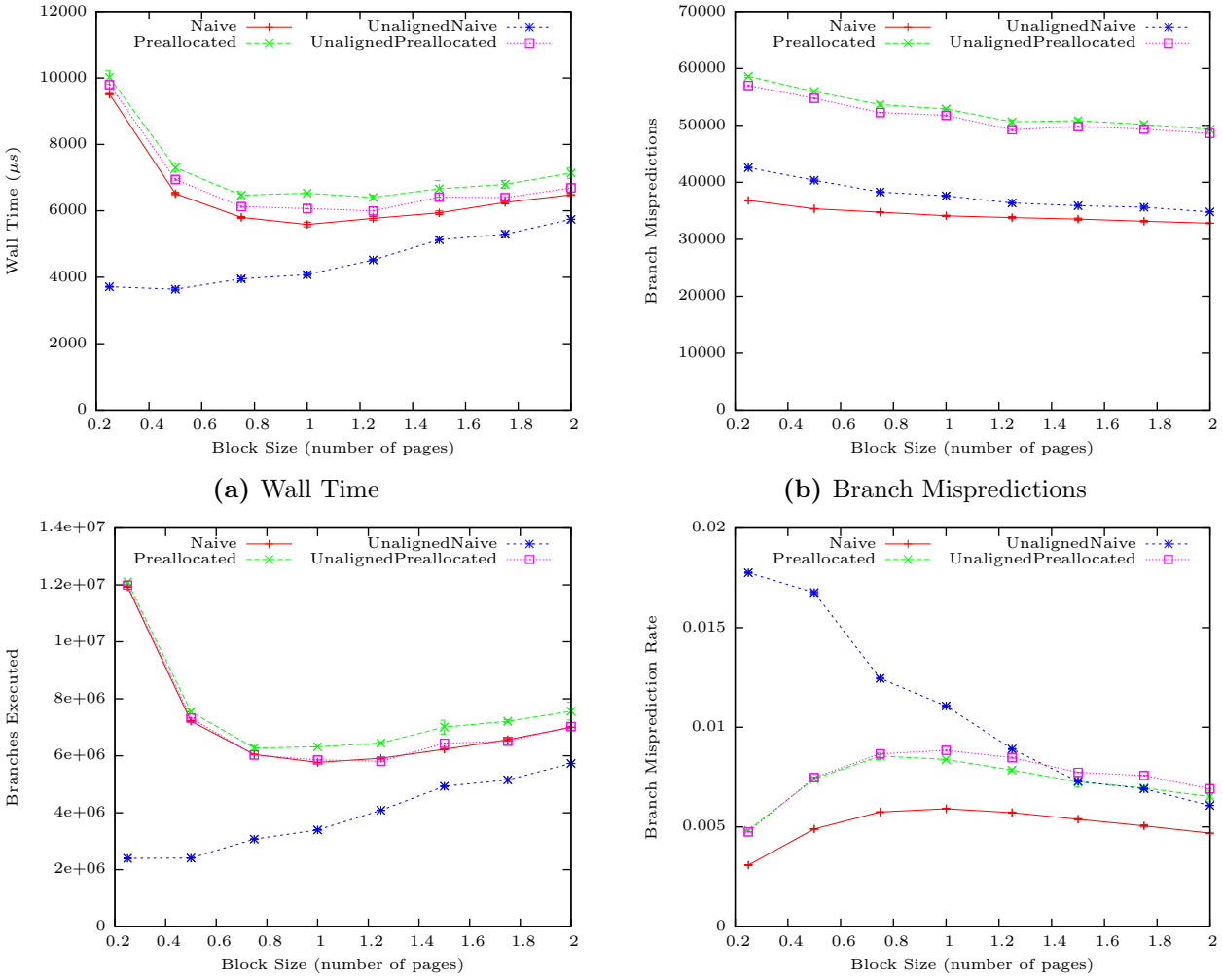
Just as with rank queries, we see higher level 2 data cache miss rate at lower block sizes in Figure 19e, and again we expect the level 1 data cache misses are the cause as we see level 1 misses matches up with level 2 hits in Figure 19b and Figure 19d.

However, unlike for the rank queries, the amount of level 2 data cache misses are not lower for smaller block sizes, in fact they are higher than at 1 page per block, where each type of wavelet tree has its minimum, with Naive having the largest drop there. We see this drop again in level 3 total cache misses in Figure 19f. For both level 2 and level 3 cache misses, we also see a dip at 2 pages per block and lesser dips at 0.5 pages and 1.5

pages per block. These sizes correspond to where blocks most often align with full pages as a block of size 2 pages will align with two pages and two blocks of size half a page will align with one page and two blocks of size 1.5 page will align with three pages.

The fact that the level 2 and level 3 performances are so near-identical can be explained by the fact that they are inclusive, meaning that everything contained in level 2 is also in level 3 and if all the cache misses in level 2 are from the prefetcher not being able to figure out what data is needed next, and it loads directly into level 2 then the level 3 cache will never have the correct data while 2 does not.

Why this results in fewer level 2 and level 3 cache misses, we do not know.



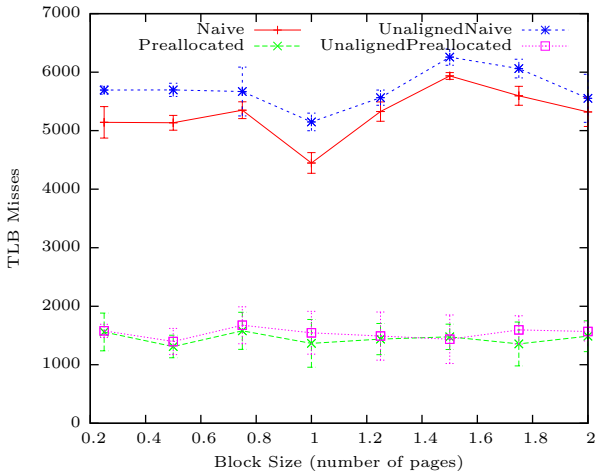
(a) Wall Time

(b) Branch Mispredictions

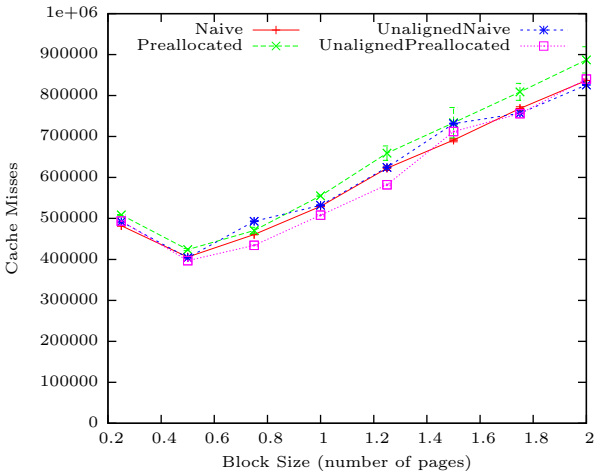
(c) Branches Executed

(d) Branch Misprediction Rate

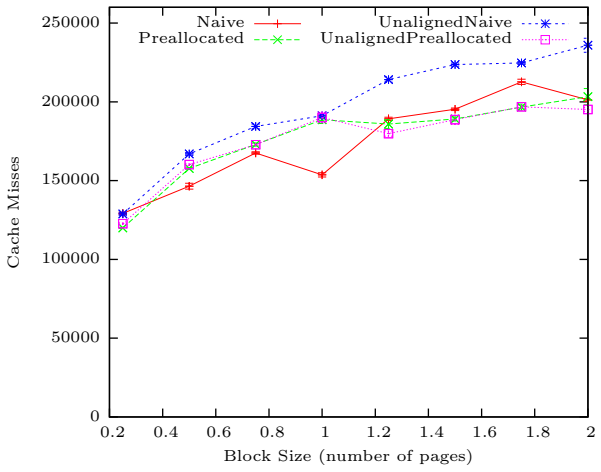
Figure 16: Various measurements of Rank queries on Wavelet Trees with Precomputed Rank Values for varying block sizes, part 1



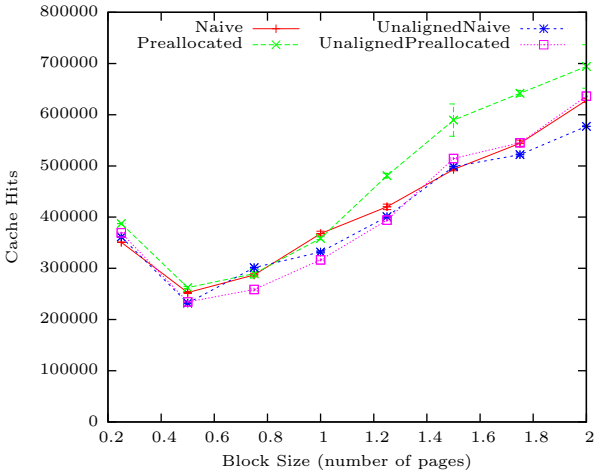
(a) TLB Misses



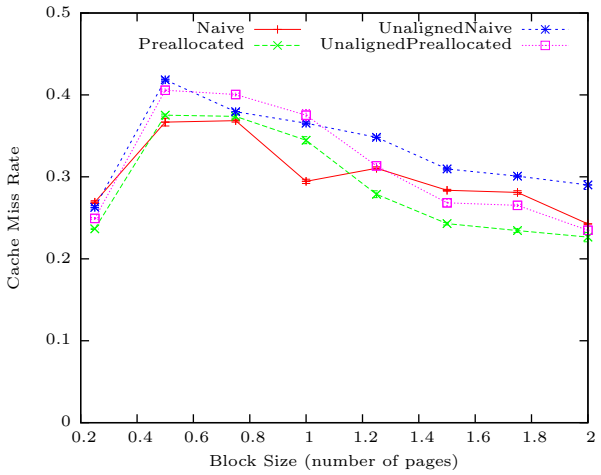
(b) Level 1 Data Cache Misses



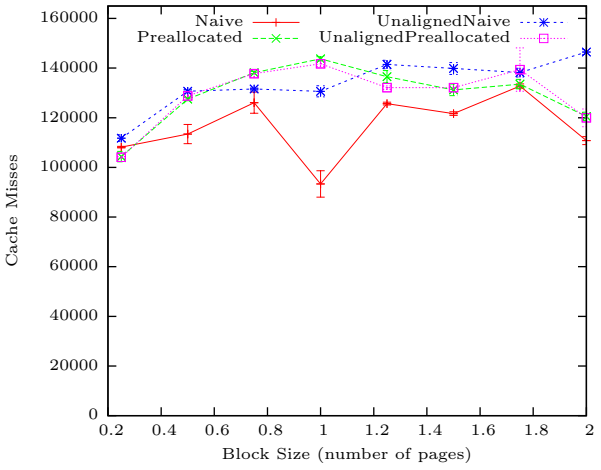
(c) Level 2 Data Cache Misses



(d) Level 2 Data Cache Hits

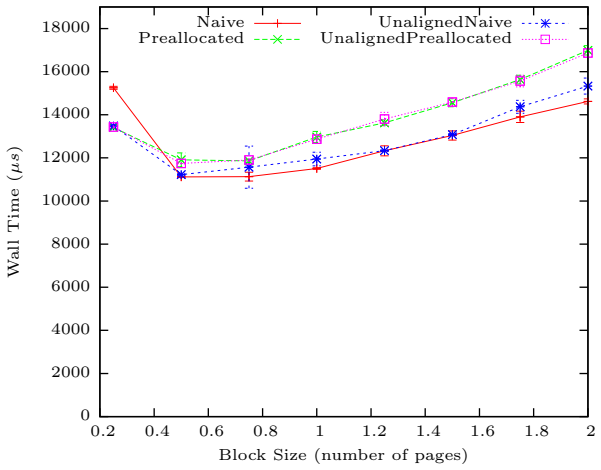


(e) Level 2 Data Cache Miss Rate

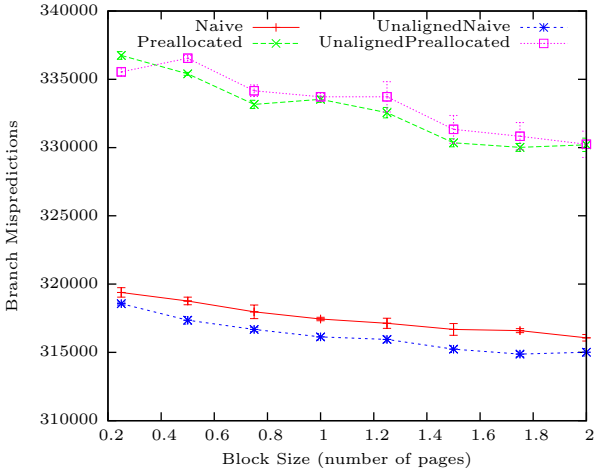


(f) Level 3 Total Cache Misses

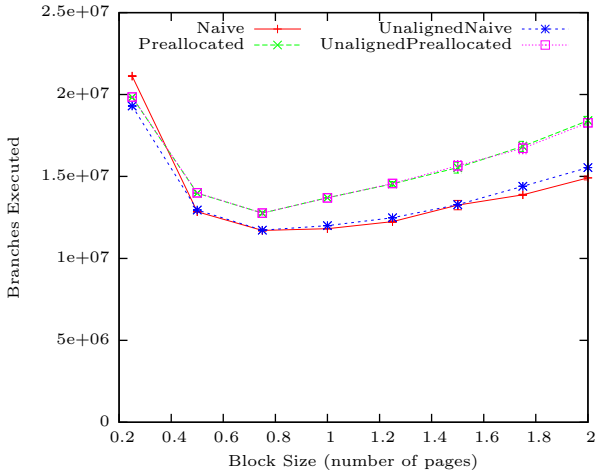
Figure 17: Various measurements of Rank queries on Wavelet Trees with Precomputed Rank Values of varying block sizes, part 2



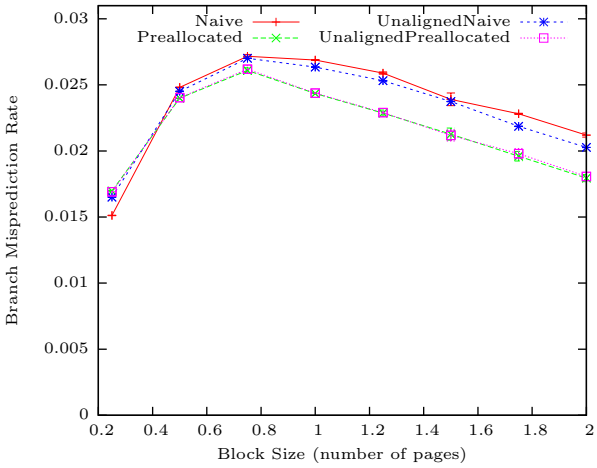
(a) Wall Time



(b) Branch Mispredictions. Notice the y-axis not starting at 0.

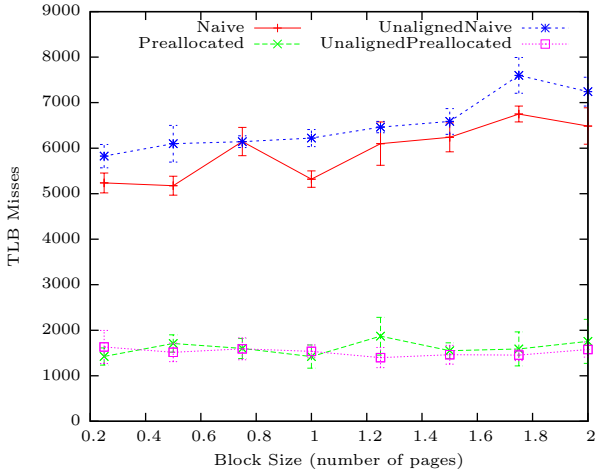


(c) Branches Executed.

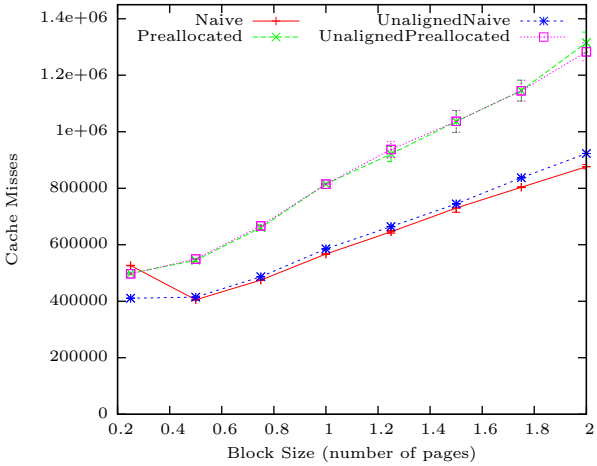


(d) Branch Misprediction Rate

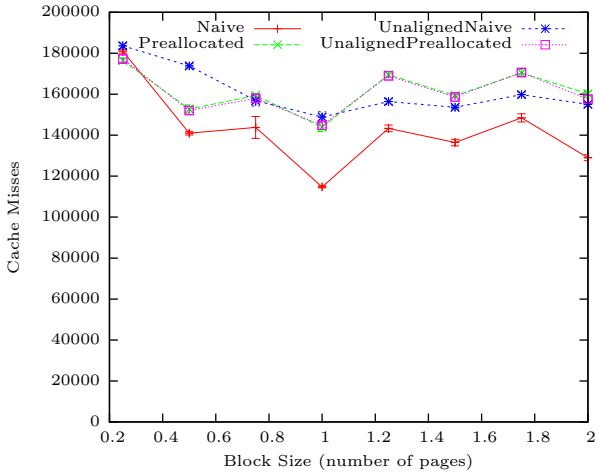
Figure 18: Various measurements of Select queries on Wavelet Trees with Precomputed Rank Values of varying block sizes, part 1



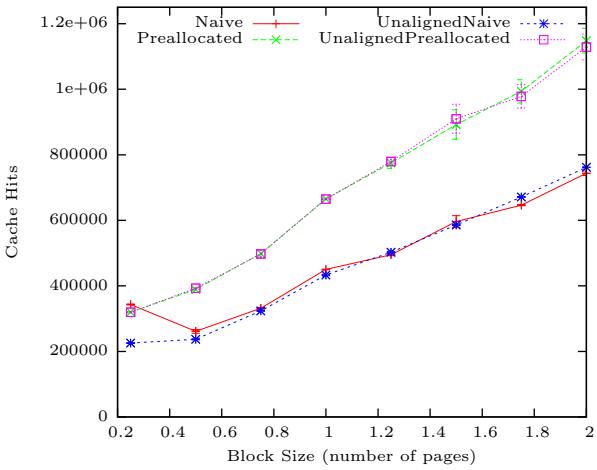
(a) TLB Misses



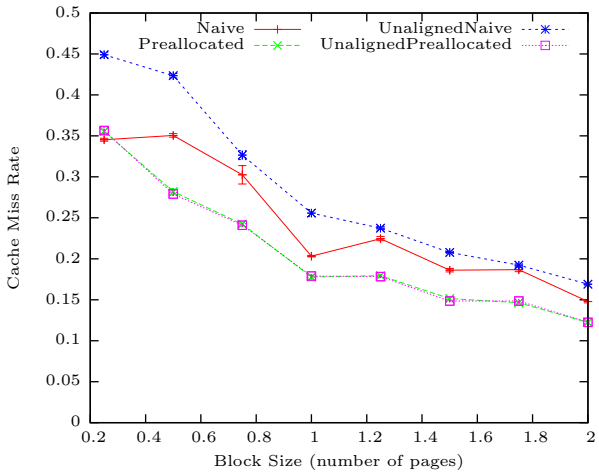
(b) Level 1 Data Cache Misses



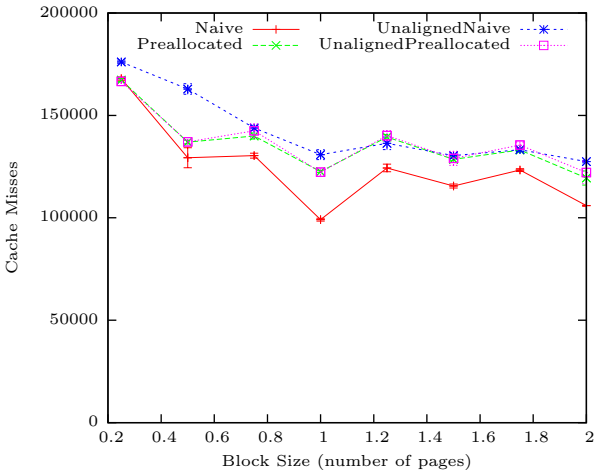
(c) Level 2 Data Cache Misses



(d) Level 2 Data Cache Hits

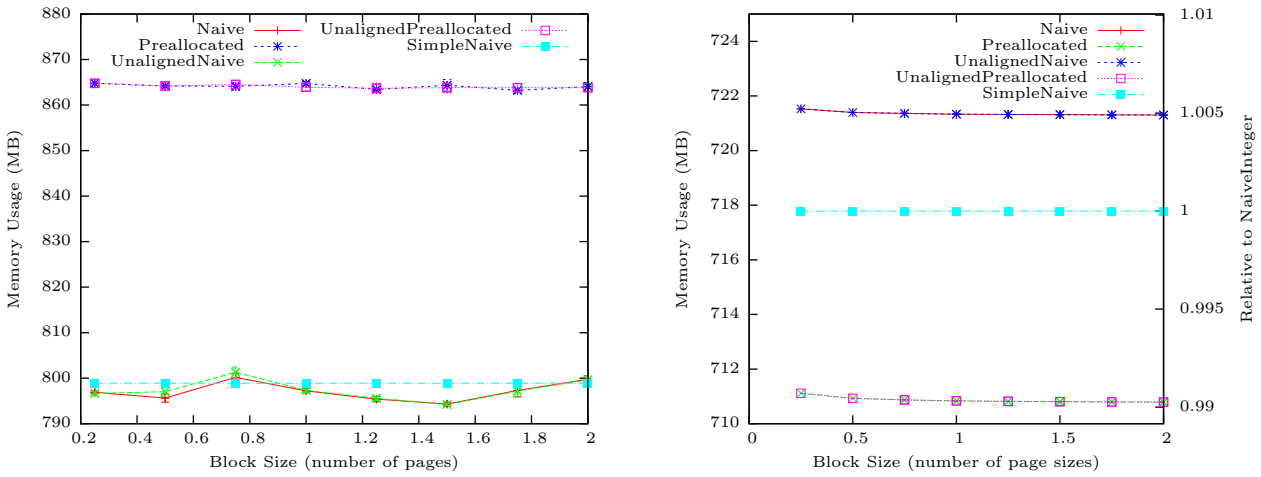


(e) Level 2 Data Cache Miss Rate



(f) Level 3 Total Cache Misses

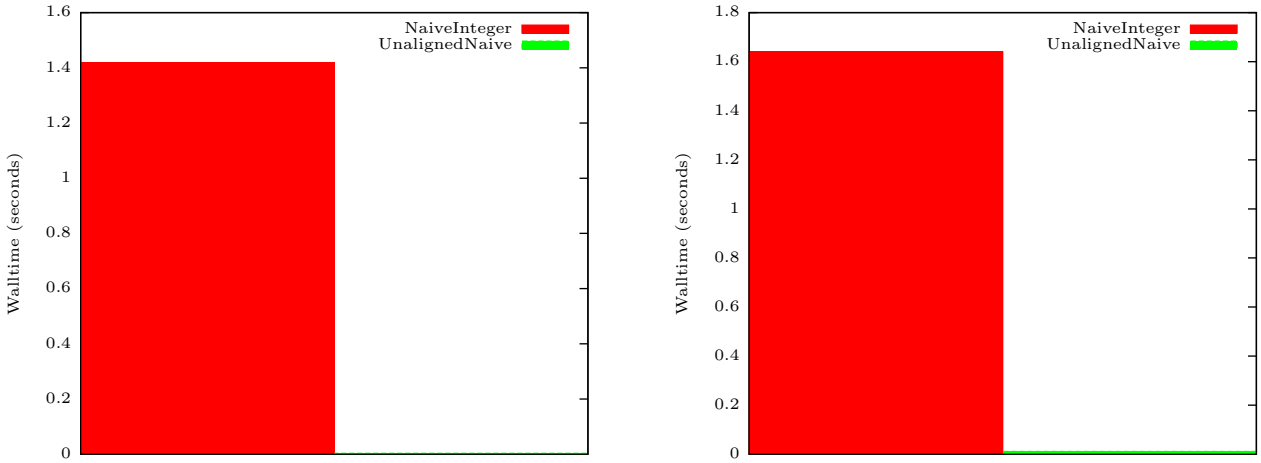
Figure 19: Various measurements of Select queries on Wavelet Trees with Precomputed Rank Values of varying block sizes, part 2



(a) Reported by PAPI

(b) Reported by Massif.

Figure 20: Difference in Memory Usage of wavelet trees with precomputed ranks of varying block size. Notice the y-axis not starting at 0 and at different values.



(a) Rank

(b) Select

Figure 21: Comparison of wall time of rank and select queries between SimpleNaive not using precomputed values and UnalignedNaive using precomputed values.

9.5.2 Memory Usage of Precomputed Rank Values

We have used both Massif, the heap profiler included in the Valgrind Suite and PAPI to record the memory usage of our programs. We focus on the output from Massif instead of PAPI, because the memory usage values we could extract using PAPI made little sense, with values indicating that the trees using precomputed values used less memory than the SimpleNaive tree with no extra precomputed values. We have plotted the values from PAPI in Figure 20a.

We do not know why PAPI gives us these nonsensical results, but it might have something to do with Linux's aggressive caching and buffering. We are not entirely sure whether `delete`ed memory might still be counted as used memory by PAPI because of such caching. Massif, on the other hand, manually counts calls to such functions as `new` and `delete` and calculates the memory usage from these, making it unaffected by any caching scheme employed by the Linux kernel. Massif is also designed for the purpose of profiling program memory usage, whereas PAPI is simply an API to access performance metrics from the underlying OS.

Massif only counts heap allocations and deallocations per default and not for stack, data, BSS or code segments. We manually tell Massif to count our stacks as well and include that in our calculations. The remaining uncounted data, BSS and code segments should not affect our results noticeably as those segments are usually tiny compared to the size our program is using. But it might be the case that it is in fact these segments causing the strange results from PAPI, and that using concatenated bitmaps uses more memory than not and storing more information in the tree somehow uses less memory, sometimes, though we find this unlikely and choose to trust Massif's output more.

Massif outputs 'snapshots' of the memory usage taken at certain points in the code where it deems them useful to the user. We have not simply used the snapshots produced automatically by Massif, but rather used the Client Request mechanism [23]of valgrind to send a `snapshot` command to the valgrind gdbserver, telling it to take a snapshot right after we have finished building the tree. The values presented in Figure 20b is thus calculated from such Massif snapshots taken right after the tree has completed building.

Looking at Figure 20b we see that the memory usage of the trees using precomputed values but not concatenated bitmaps, Naive and UnalignedNaive use more memory than the tree not using precomputed values, SimpleNaive, but only about 0.5 %. We also see that memory is indeed saved by using the concatenated bitmaps in Preallocated and UnalignedPreallocated, though only about 1.5 % compared to the other trees using precomputed values.

Our attempts to reduce the memory usage by concatenating the bitmaps seems to have succeeded, but looking back at the running times for the rank and select queries in Figure 31a and Figure 31b, we suspect the few percentage reduction in memory usage is not worth the massive decrease in rank query performance and the slight decrease in select query performance.

[23]`http://valgrind.org/docs/manual/manual-core-adv.html`

9.5.3 Improvement of using precomputed values

We have found that our UnalignedNaive precomputed tree using block size = 0.5 page size is the fastest for rank queries for input size $n = 10^8$ characters, and about as fast as the Naive tree for select queries. We therefore consider UnalignedNaive at block size 0.5 page size to be our best wavelet tree implementation so far, even though it uses more memory than SimpleNaive or the Preallocated variants.

We have compared the running time of 1000 rank and select queries for UnalignedNaive and SimpleNaive in Figure 21. The wall time for rank queries on the UnalignedNaive tree is only 0.27 % of the wall time of rank queries on the SimpleNaive tree as can be seen in Figure 21a. The wall time for select queries on the UnalignedNaive tree is only 0.73 % of the wall time of select queries on the SimpleNaive tree as can be seen in Figure 21b.

We can see that it is a great improvement to use precomputed rank values in both rank and select queries, with more gain for rank.

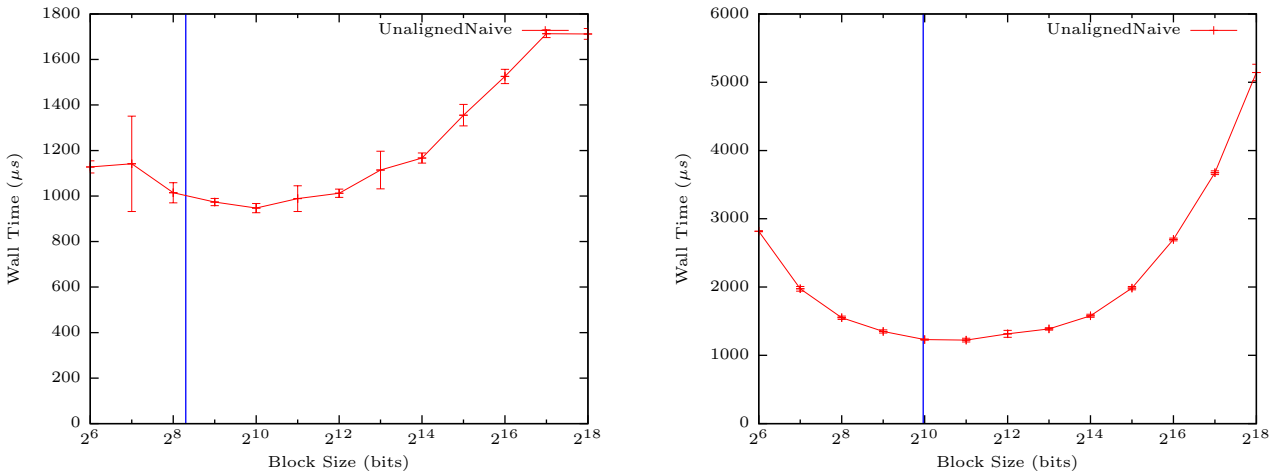
9.5.4 The Dependence of Optimal Block Size on Input Size

We have run our experiment for rank queries on the UnalignedNaive wavelet tree using varying block sizes, for 4 different input sizes, $n = [10^5, 10^6, 10^7, 10^8]$, to show that the optimal block size b depends on n, and to show that the optimal block size b is less that the theoretically optimal b for the root bitmap, when using a fixed block size throughout the wavelet tree. We did not run this experiment for select queries, because of problems choosing appropriate query parameters for the smaller input sizes, and because we felt little additional valuable information would be gained.

In Figure 22 we have plotted the wall times for various block sizes for four different values of n. The x-axis is logarithmic as we tested for block size values as powers of 2, to reach a wide range of block sizes without having to run hundreds of tests yet still have several tests at low values.

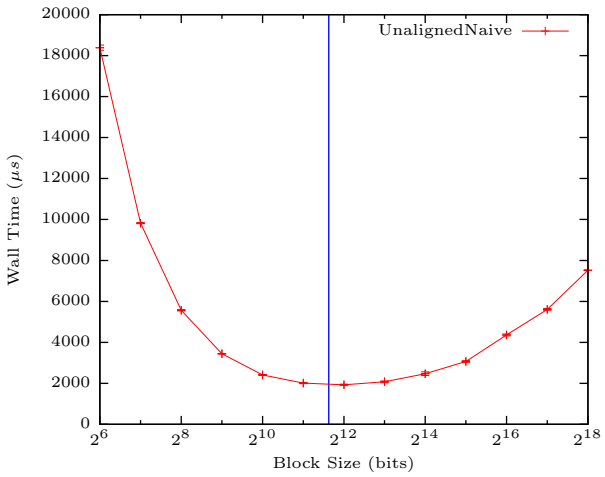
For all four graphs in Figure 22 we can see that the performance at the theoretical optimal block size for the root bitmap at $\sqrt{n}$ is good, and close to the minimum in wall time. Therefore, the wavelet tree using precomputed rank values in blocks should compute its block size based in the size of the input for better performance. A further improvement might be to compute the block size for each node of the tree individually. We have, however, not done this in our implementation, because of some problems with the implementation and time constraints.

Surprisingly, we find that the minimum in wall time is at a block size not below the theoretical optimal block size, but instead slightly above it if anything at all. We had expected that the other bitmaps, smaller than the root bitmap and therefore having smaller optimal block sizes, would have skewed the optimal block size downward. The difference in performance between the theoretically optimal block size of $\sqrt{n}$ and the measured optimal block size is small, especially for the larger input sizes, and using a block size of $\sqrt{n}$ would be a sufficient optimization in most cases.

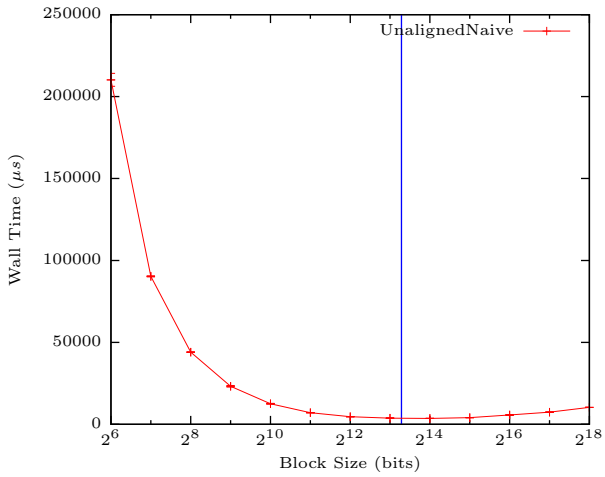


(a) $n = 10^5$

(b) $n = 10^6$



(c) $n = 10^7$



(d) $n = 10^8$

Figure 22: Walltimes for varying block sizes are four different input sizes. The vertical blue line is at $\sqrt{n}$, the theoretical optimal block size for the root bitmap. Notice that the x-axis is logarithmic.

10 Precomputed Cumulative Sum of Binary Ranks

We have found that using precomputed rank values is a great improvement to the running time of both rank and select queries, though with a higher gain for rank queries. It works so well, because it allows the algorithms to skip most of the bitmaps, only directly accessing them near the position that was queried for in case of rank queries and near the sought-after occurrence in the case of select queries, and relying on the precomputed values for the rest of the bitmap.

It is still however necessary to iterate through the precomputed values. Most of the time the algorithms are interested in the rank value at some position inside a bitmap and it is the rank from the beginning of the bitmap to the position and rarely just the rank of that particular block. Therefore it might be possible to save a number of instructions by not iterating through the precomputed values if the precomputed values were already this cumulative sum of rank values through the bitmap.

We implement this based on UnalignedNaive, again testing the performance for various block sizes to fin the optimal block size and then compare that performance to the UnalignedNaive wavelet tree from Section 9.5.1.

10.1 Advantages of Cumulative Sum

As previously mentioned, the rank and select query algorithms do not actually need the rank values of individual blocks, but rather the cumulative sum rank value from the beginning of the bitmap to some position. If we instead implement the precomputed values as being the cumulative sum of rank values of each block from the beginning of the bitmap up to and including the block corresponding to the precomputed value, we can save a lot of precomputed value lookups in the rank and select queries.

Calculating the cumulative rank sums during the construction does not require much more computation. It can e.g. be done by a single sweep through the precomputed values vector after having computed the entire bitmap, adding each precomputed value to the next in the vector.

Rank queries will benefit from the precomputed values being cumulative sums because they can do a single lookup of the precomputed value corresponding to the block covering the queried-for position. The need to calculate rank by using `popcount` within a single block remains unchanged. This means that the required work per level of the tree changes from $O(\frac{n}{b} + b)$ to $O(b)$ because binary rank becomes an $O(b)$ operation, making the total work required for a rank query $O(b \log \sigma)$.

Select queries should also see some benefit. Previously, the select query would iterate through the precomputed values and sum them up, looking for when it surpasses the sought-after occurrence, and then calculate the position within a single block using `popcount` and manual counting of bits within a single word. Using cumulative precomputed rank values, the select query is able to use binary search on the precomputed value vector to find the word wherein the occurrence is. Using `popcount` within a block and manual counting within a word still remains unchanged. This makes the previously

required work per level change from $O(\frac{n}{b} + b)$ to $O(\log \frac{n}{b} + b)$, making the total work required for a select query $O((\log \frac{n}{b} + b) \log \sigma)$.

In Section 10.5.2 we test what block size achieves the best running time for rank, select and branchless select. We test rank for low block sizes since $O(b \log \sigma)$ indicates that the lower b is, the faster the running time is. The running time for select can also be written as $O(b \log \sigma + \log \frac{n}{b} \log \sigma)$ and comparing it to the running time of rank we can see that select has an extra term: $\log \frac{n}{b} \log \sigma$. The effect of this extra term is that our expected optimal block size is higher for select than for rank, because this term decreases as b increases.

10.2 Disadvantages of Cumulative Sum

The memory analysis remains the same as before, at $O(n \log \sigma + (\sigma + \frac{n}{b} \log \sigma) \cdot ws)$ bits, because it is still one number stored per block. However, in practical terms, the precomputed values are no longer limited in value size by the block size but rather the bitmap size, as the last value in the precomputed rank value vector could potentially become as large as the bitmap is long. Storing the cumulative sums will then require more bytes per value and thus use more space in the end.

The bitmap size is limited by the input string length and so, for our choice of input string with length 10^8 characters, each precomputed value must be able to store a value up to 10^8. It takes at least 28 bits to store the value 10^8, because $2^{27} < 10^8 < 2^{28}$. Because the value types supported by x86 and C++ must be byte (8-bit) aligned and use a number of bytes that is a power of 2, the smallest type we can use is the 4-byte type `unsigned int` capable of storing values up to 2^{32}. This means the vector, instead of holding 2-byte `unsigned short int`s, must hold 4-byte `unsigned int`s, doubling the space required to store the precomputed values. We expect this increase in memory usage to be tiny, as it is another 2 bytes per block, of which there are $\frac{n}{b}$ per layer of the tree, of which there are $\log n$. So with our input string of length $n = 10^8$ and a blockSize of $2^{10} = 1024$ bits, we expect an extra memory usage of about 634 kB:

$$\text{Memory usage} = 2 \cdot \frac{10^8}{1024} \cdot \log 10^8 \approx 648,814 \text{ bytes} \approx 634 \text{ kB}$$

We have already seen that the UnalignedNaive wavelet tree for this input size and block size and an alphabet size of 2^{16} uses about 721 MB of memory, so another few hundred kilobytes is barely worth mentioning. We will see in our experiments how much actual memory is used and whether the difference in running time can make up for the increase in storage space required.

10.3 Optimal Block Size

Like when using non-cumulative precomputed rank values, the block size can affect the performance. But, unlike using non-cumulative precomputed rank values, when using cumulative precomputed rank values the optimal block size b for rank queries is not affected by the input size n. This is because the rank algorithm no longer has to linearly

scan through the precomputed rank values, but can perform a single lookup before using popcount. This is also reflected in the running times of $O(\frac{n}{b}+b)$ for non-cumulative and $O(b)$ for cumulative, as there is no n term in the cumulative running time.

From the theoretical running time of $O(b)$, we expect the optimal block size to be small. However, any block size below the size of word `popcount` operates on, 64 bit for our machine, will likely not be any improvement, as using `popcount` to calculate the rank within that word takes constant time. The only exception, we expect, is if a block size of 1 bit was used and the algorithm modified to just use that precomputed value and not use `popcount`, which corresponds to precomputing the answer to every possible rank query, and using a lot of memory in the process. We will not be testing this, though we will do experiments for block sizes smaller than 64 bit.

For select queries, there is still a dependence on n, as it has to perform a binary search over the precomputed rank values, and that is reflected in the theoretical running time of $O((\log \frac{n}{b} + b) \log \sigma)$.

10.4 Select Queries with less branching code

When implementing the select query for the cumulativeSum wavelet tree, we realized it included a lot of `if/else` branches that could be difficult to predict by the branch prediction unit. We anticipated that we might improve upon the query by eliminating as much branching code as possible. That is, reduce the number of `if/else` statements, `while`-, and `for`-loops in the code and instead replace them with "clever" arithmetic operations achieving much of the same.

One large disadvantage of this approach was that it resulted in a binary search that did not terminate early if the correct block was reached, but would instead always jump and do a lookup $\log(\mathit{blocksInNode})$ times for each node. Many of the later jumps that would be skipped by terminating early lie close in memory and with high probability exist in the same cacheline and thus be fast to lookup. This fact, combined with a reduction in branch mispredictions could make this "branchless" version faster.

Based on experiments we found that (Select) was slower when using the "branchless" approach than just using the simple approach (see Section 10.5.4). When we realized this we attempted to combine the two approaches to get the best from both: early termination from the simple approach and less branching code meaning fewer branch mispredictions from the "branchless" approach. However, whatever we tried, it always seemed to be slower than the simple approach, and so we stopped trying to combine the two and there a no experiments for a combined approach.

It makes sense that the branchless select is slower than the branching version of select with mispredictions. According to Agner Fog[24] the branch misprediction penalty in the Ivy Bridge Architecture is at least 15 clock cycles. If a branch is very hard to predict by going one way half the time and the other way the other half, it would be mispredicted about 50 % of the time, making it cost on average $1 + \frac{15}{2} = 8.5$ clock cycles if we assume that a correctly predicted branch costs 1 clock cycle. This means that our alternative

[24]Section 3.7 in `http://www.agner.org/optimize/microarchitecture.pdf`

method of computing something without using branches must cost less than 8.5 clock cycles to be an improvement. Looking at our code, the alternative method where we have attempted to reduce branches could easily cost much more than 8.5 clock cycles. This also fits with our experimental data (see Section 10.5.4 and Figure 26) because both cache misses, branch mispredictions and TLB misses are smaller for "branchless" select than for the branching version of select, yet the wall time is higher. An increased amount of cycles can explain why this method results in a slowdown while reducing hardware based penalties.

10.5 Experiments

With the following experiments we want to test whether the changes described in the previous section achieves any improvements in practice. We want to know what effect the changes have on the amount of hardware penalties incurred and try to explain why. We show the trade-off between build time, memory usage and running time of queries. We test tree construction and rank and select queries for different block sizes of cumulative sums of precomputed rank values to find the one achieving the best running time. We compare rank and select for UnalignedNaive vs. using cumulative sum and show how their running time and hardware penalties differ.

10.5.1 Build Time And Memory Usage For Various Block Sizes

In Figure 23 we have plotted the wall time and memory usage of building the UnalignedNaive and CumulativeSum Wavelet Trees. In Figure 23a we can see that it takes slightly longer to build the tree when we have to calculate the cumulative sum across the precomputed values we store. The difference at 2^{10} is 0.49 seconds, which is a 3.33 % increase from UnalignedNaive to CumulativeSum, and for other block sizes similar differences are found.

In Figure 23b we can see that, as expected, CumulativeSum takes more memory, but only significantly so when using block sizes less than 2^8 bits (8 bytes). At the lowest block size we tested for, 2^3 bits, we see a massive increase in memory usage. For CumulativeSum, the memory usage at a block size of 2^3 bits is about double that at a block size of 2^8 bits.

If we look closer at the raw data at block size 2^{10} for which we calculated an expected extra memory usage of 634 KB., there is an increase of 350 KB in memory usage when storing the cumulative sum, which constitutes an increase of about 0.38 %, and is even less than what we expected and is negligible compared to the expected increase in running time.

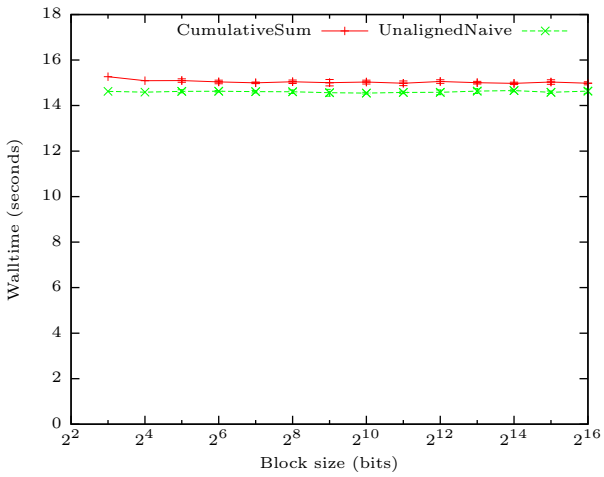
10.5.2 Optimal Block Size For Rank And Select

We have made tests of the running time of Rank, Select, and SelectBranchless queries on wavelet trees of varying block sizes from 2^2 to 2^{16} bits. The test results are shown in Figure 24.

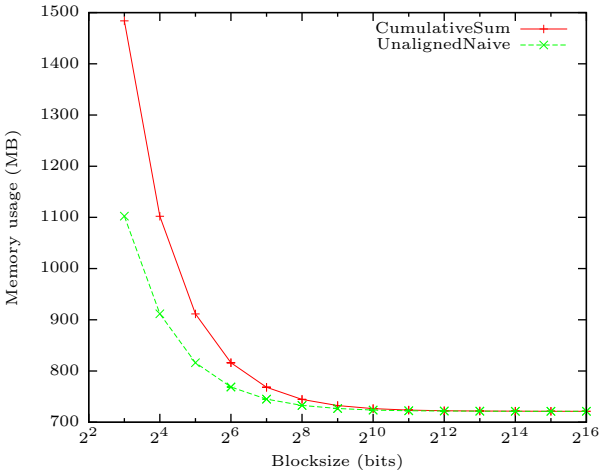
From Figure 24a we observe that the best running time of rank queries is achieved using a block size of $2^6 = 64$ bits. The blue line indicates the theoretically best block size of 64 bit as explained in Section 10.3 and now confirmed by this test.

Select achieves the best running time with a block size of $2^{11} = 2048$ bits which can be observed in Figure 24b and the branchless version achieves the best running time using a block size of $2^{10} = 1024$ bits as seen in Figure 24c The found block sizes fits with the theoretical Big-O analysis. Rank is best with a small block size and select is also better with a relatively small block size that is larger than for rank.

In a realistic use case one would want to build a single tree using one block size and do rank and select on that tree and not have two trees with different block size, one for rank and one for select. From our experiments, a block size of $10^{10} = 1024$ bits seem to be the best choice when using an input string of 10^8 characters. It has close to optimal query running time for both rank and select, and only uses about 0.38 % more memory.

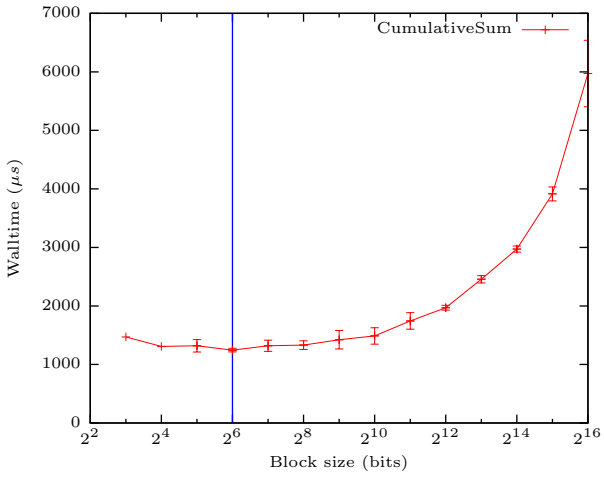


(a) Wall Time

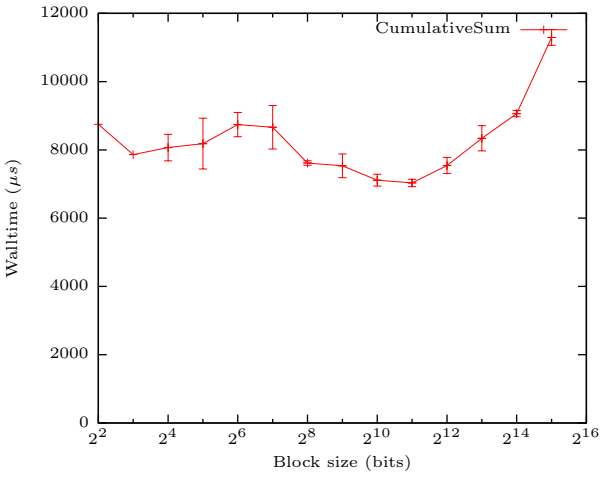


(b) Memory Usage. Note that the y-axis does not start at 0.

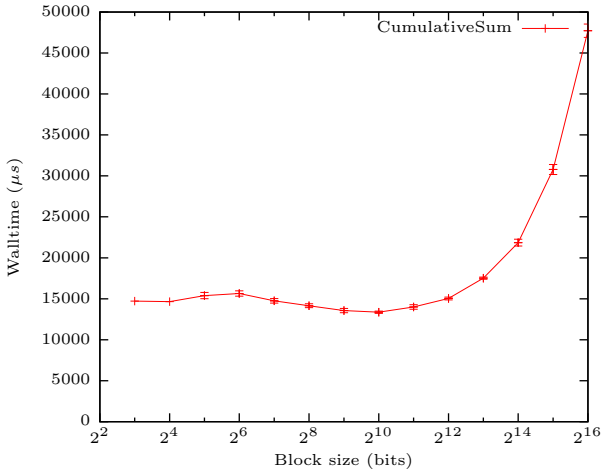
Figure 23: Measurements on Building the UnalignedNaive and CumulativeSum wavelet trees. The x-axis (block size) is logarithmic.



(a) Rank. Blue line marks expected best block size.



(b) Select.



(c) Branchless Select.

Figure 24: Running times of CumulativeSum rank and select for varying block sizes with $n = 10^8$ characters. The x-axis (block size) is logarithmic.

10.5.3 Rank Queries

In Figure 25 we have plotted various measurements for Rank queries on the UnalignedNaive and CumulativeSum trees. In Figure 25a we can see that storing and using the cumulative sum of rank values instead of the rank values for each block improves the running time of rank queries. UnalignedNaive spends 4.05 milliseconds on 1000 queries, where CumulativeSum spends 1.43 milliseconds, a reduction in running time of 64.7 %.

Looking at Figure 25b, Figure 25d we can see that the tree using cumulative sums has much fewer branch mispredictions but a higher misprediction rate, which can be explained by the fact that fewer conditional branches are executed overall during rank queries as seen in Figure 25c. The decreased amount of branch mispredictions can be explained by the removal of a for-loop in CumulativeSum that iterated over the precomputed values, summing them up to calculate the rank, instead replacing it with a single lookup of a precomputed value.

In Figure 25e we see that the CumulativeSum tree has slightly more Translation Lookaside Buffer Misses than UnalignedNaive but not lot, so the amount of TLB misses are not reduced when using a cumulative sum.

In Figure 25f, Figure 25g, and Figure 25i we can see that rank queries on the CumulativeSum wavelet tree has a much better level 1 cache, level 2 and 3 cache performance because of the decreased amount of cache misses. Cache misses are reduced and this helps to improve the CumulativeSum rank running time because fewer cache lookups are needed.

In Figure 25h we can see that the amount of level 2 cache hits decrease significantly when using cumulative sums of the precomputed values. The explanation for the decrease in level 2 cache hits might lie in the reduction of level 1 cache misses as seen in Figure 25f, like results from previous experiments. The reduction in level 2 cache hits is mainly the amount of cache lookups that the level 1 cache instead was able to handle. The reduction of level 1 cache misses is on average 319 996 and the reduction in level 2 cache hits is on average 213 818 which seems to support this. The level 2 cache miss rate (not shown) is therefore somewhat misleading as it would suggest a worse cache performance where the truth is that CumulativeSum has a much better cache performance, having much fewer level 1 cache misses, which helps to explain why the rank queries are faster.

10.5.4 Select Queries

In Figure 26 we have plotted the same measurements as in Figure 25, but for Select queries, including our "branchless" variant of the cumulativeSum select query.

In Figure 26a we can see that storing and using the cumulative sum of precomputed rank values is also an improvement for select queries, with a reduction in wall time of 40 %. Our "branchless" approach is also faster than not using the cumulative sum, but much slower than the simpler approach only achieving a wall time reduction of 18 %.

Looking at Figure 26c we can see that both approaches using the cumulative sum executes much fewer conditional branches, which could be caused by using the binary search instead of having to iterate through every precomputed value from the beginning

of the bitmap to the position where the sought-after occurrence lies. The "branchless" select also executes fever branches than the branching version of select. The difference is not large though, which could mean that most of the branches are from traversing the tree rather than computing the binary select on the bitmap.

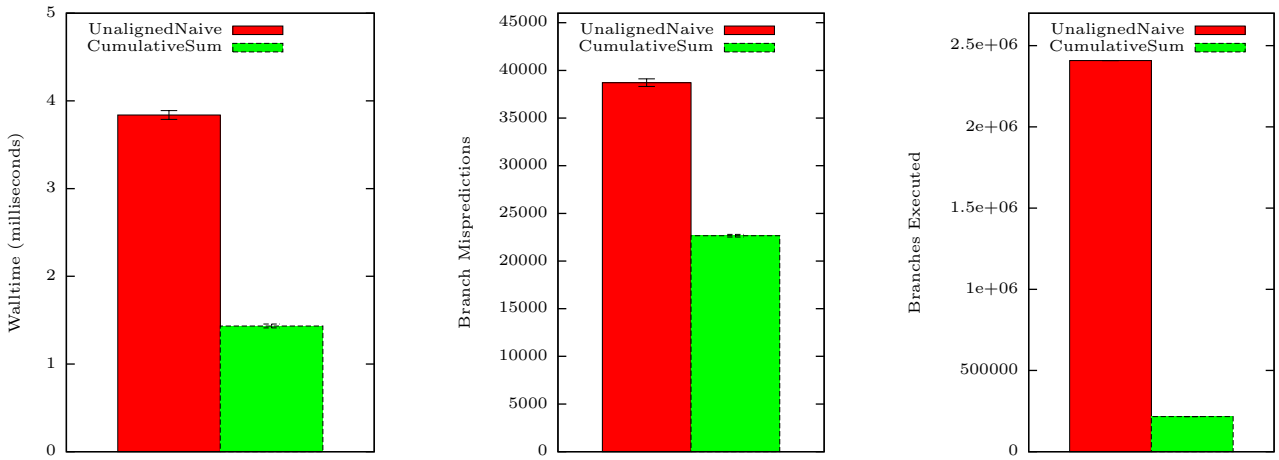
In Figure 26b and Figure 26d we can see that the branching approach using cumulative sum has, as expected, more branch mispredictions and a higher branch misprediction rate than both of the others. The additional number of branch mispredictions contribute about 936 255 extra clock cycles compared to UnalignedNaive which, assuming 15 clock cycles per misprediction, is only 4.5 % of the total number of clock cycles, which is 20 954 197, used in the cumulative sum branching select query.

In Figure 25e we can see that the branching approach also has more TLB misses than the others, yet this still has not made it slower than the others.

In Figure 26f, Figure 26g, and Figure 26i we can see that using a tree with cumulative sum again has better level 1, level 2 and 3 cache performances than UnalignedNaive. The "branchless" approach has the best level 1 cache, level 2 and 3 cache performances as was also the case for rank. We see a reduction in level 2 cache hits as for rank in Figure 26h from UnalignedNaive to the two CumulativeSum approaches. This rediction can again be explained mainly by the reduction in level 1 cache misses as seen in Figure 26f. Level 2 cache misses are a little higher for branching CumulativeSum than for UnalignedNaive. The "branchless" select reduces the hardware penalties more that branching select but it is still slower because requires more cycles to achieve "branchless" select and the hardware penalty reduction does not make up for this increase.

In the end we can confirm based on our measurements that it can be explained why Rank and Select is faster for CumulativeSum than for UnalignedNaive. We feel the performance gain for queries by far make up for the small increase in time and memory usage when building the tree.

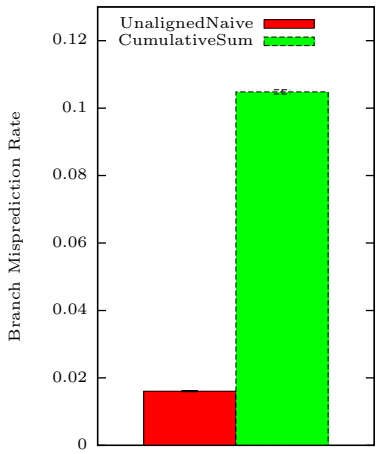
CumulativeSum is already theoretically better than UnalignedNaive (see Section 10.1) because of the cumulative sums allowing binary rank to be computed in $O(b)$ time. The tests show the practical effect of this theoretical improvement and confirms that the improvement also works in practice.



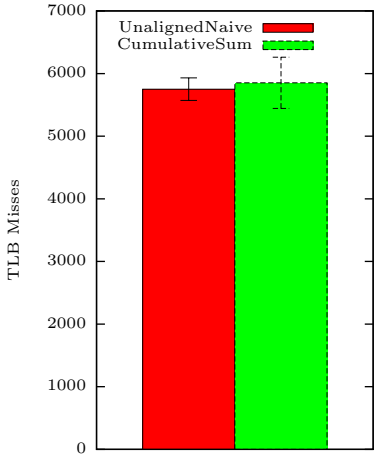
(a) Wall Time

(b) Branch Mispredictions

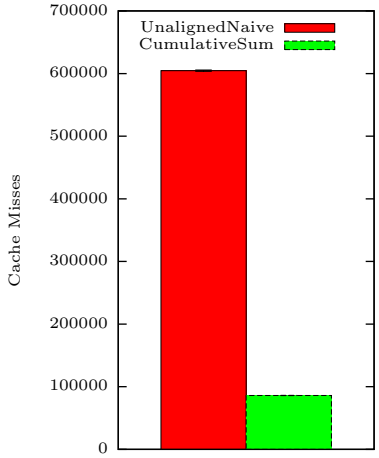
(c) Branches Executed



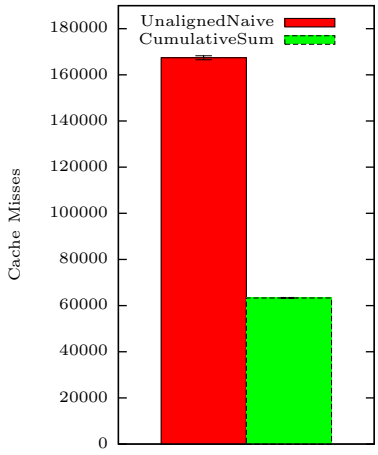
(d) Branch Misprediction Rate



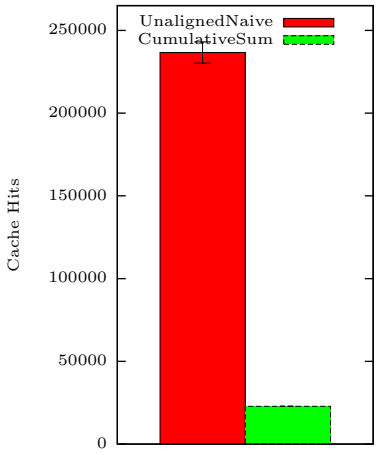
(e) TLB Misses



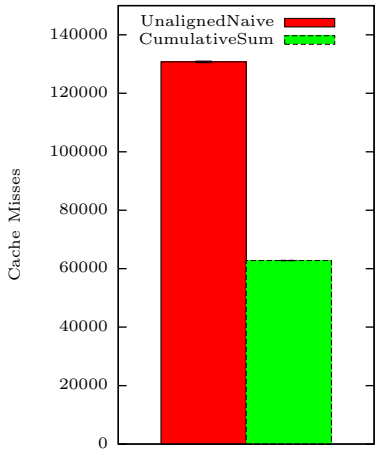
(f) Level 1 Cache Misses



(g) Level 2 Cache Misses

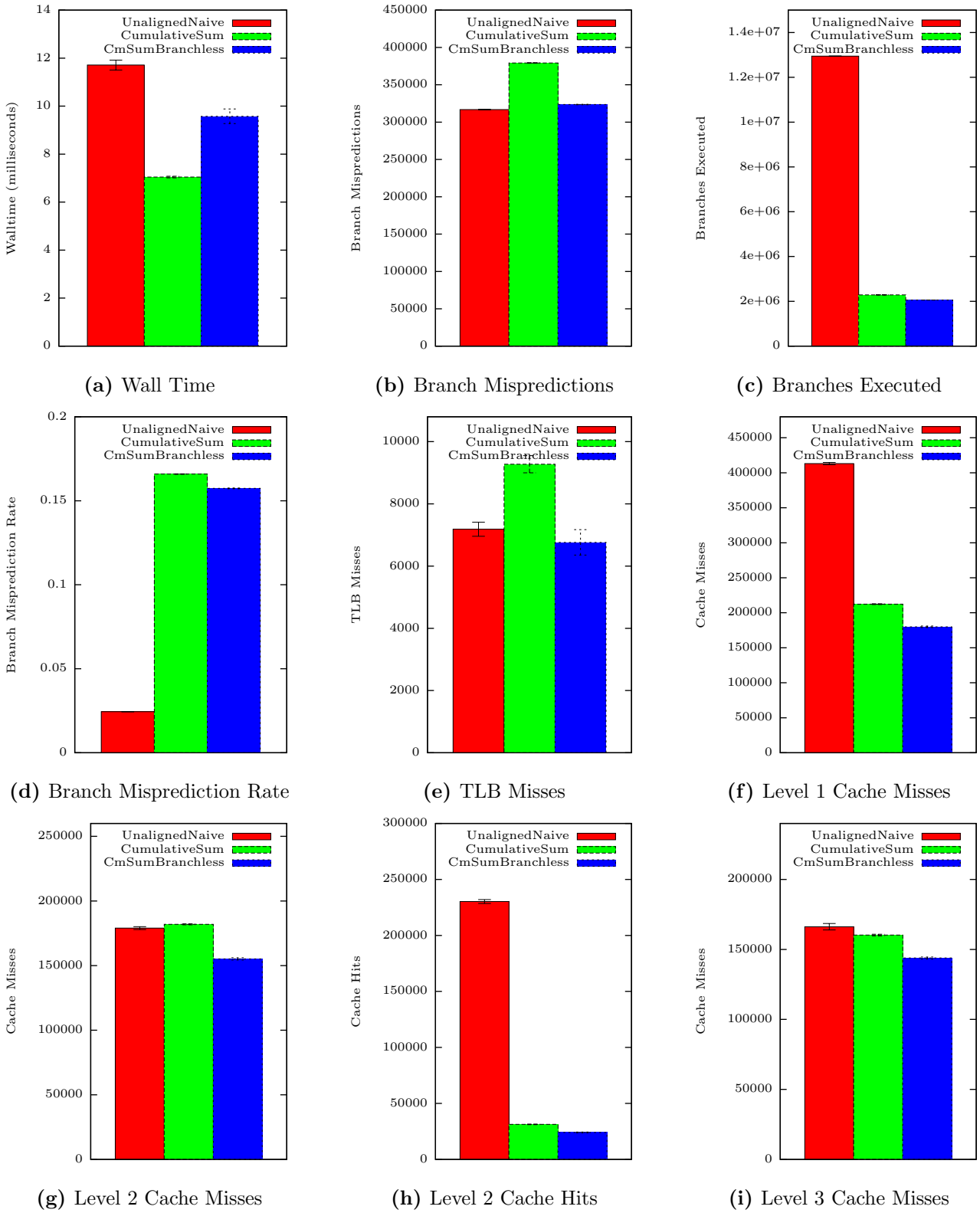


(h) Level 2 Cache Hits



(i) Level 3 Cache Misses

Figure 25: Measurements on Rank Queries on the UnalignedNaive and CumulativeSum Wavelet Trees. Part 1.



(a) Wall Time **(b)** Branch Mispredictions **(c)** Branches Executed

(d) Branch Misprediction Rate **(e)** TLB Misses **(f)** Level 1 Cache Misses

(g) Level 2 Cache Misses **(h)** Level 2 Cache Hits **(i)** Level 3 Cache Misses

Figure 26: Measurements on Select Queries on the UnalignedNaive and CumulativeSum and CumulativeSum-Branchless Wavelet Trees. Part 1.

11 Cumulative Sum with Controlled Memory Layout and Skew

In this section, we describe our attempt to improve the query times for the wavelet tree by controlling the memory layout and skewing the tree. Skewing the tree means that we force it to be unbalanced with a bias to one side. Brodal et al. [13, Abstract] showed that skewing a binary search tree could reduce the amount of cache misses and branch mispredictions considerately. Enough, in fact, to increase the speed of searching the tree manyfold, even though the skewing increased the depth of the tree structure.

To reduce cache misses by skewing the tree we must control the memory layout, because by skewing the tree to the right, we increase the likelihood of a traversal similar to a depth-first-search going down the right side first (DFSr). So we want to place the data in memory so that a DFSr traversal through the tree would result in sequential address accesses. Allocating memory dynamically as we go might produce a similar layout and controlling the memory may not lead to increased performance, but it is the only way to ensure the memory layout is as we want it.

Skewing the tree has the disadvantage of increasing the height, or depth, of the tree. J. Nievergelt and E. M. Reingold [22] defined the height for a skewed binary tree to be at most:

$$h_{max} = \frac{\log(m+1)}{\log \frac{1}{1-\alpha}} ,$$

where m is the total number of nodes in the tree, that is $2\sigma - 1$ in our wavelet tree, and $\alpha = \frac{1}{skew}$ and *skew* is the skew parameter, see Section 11.2. Together this makes the height at most

$$h_{max} = \frac{\log(2\sigma)}{\log \frac{skew}{skew-1}} .$$

Which, when the tree is balanced ($skew = 2$), makes the height at most $h_{max} = \log \sigma + 1$ which agrees with our definition of the height $h = \lceil \log \sigma \rceil$.

Let us analyse the theoretical worst case running time of constructing and querying a balanced wavelet tree vs. a skewed wavelet tree. Constructing a balanced wavelet tree takes $O(n \log \sigma)$ time because the height of the tree is $\lceil \log \sigma \rceil$ and there are n elements in each level. When skewing the tree the height of the tree becomes as defined above and the construction time becomes $O(n \cdot h_{max})$.

The query time for rank on a balanced wavelet tree is $O(b \log \sigma)$ and for select $O(b \log \sigma + \log \frac{n}{b} \log \sigma)$. In the skewed version of the tree the rank query time then becomes worst-case $O(b \cdot h_{max})$. The query time for select becomes $O(b \cdot h_{max} + \log \frac{n}{b} \cdot h_{max})$. The memory usage becomes $O(n \cdot h_{max} + (\sigma + \frac{n}{b} \cdot h_{max}) \cdot ws)$ bits.

From the theoretical analysis of construction time and query time of a skewed wavelet tree it is theoretically not an improvement to skew the tree. Skewing the tree can however reduce branch mispredictions, as shown by Brodal et al. [13, Abstract]. It does so by giving the branch in the direction the tree is skewed a much higher probability of being the correct than the other, which enables the branch prediction unit to predict correctly more often. Skewing the tree can also reduce cache misses by increasing the probability

that the next piece of memory the algorithm accesses is already loaded into a cacheline by the time it is accessed because of prefetching.

The algorithms for construction and queries in this implementation remain mostly the same as before in CumulativeSum, but with modifications to handle a controlled memory layout and a skew of the tree.

11.1 Prefetching

Prefetching is a feature of the CPU whereby it can fetch other parts of the memory into cachelines even though it was not requested yet, if it expects it will be requested soon, to avoid having the program waiting for this fetching. See also Section 5.1. In more advanced versions, it can even look at the access into memory of the running program and try to determine a pattern and prefetch memory according to this pattern. Looking at the Intel Optimization Manual [23] for our architecture[25] we find that it has streaming prefetchers loading into level 1, level 2, and level 3. The streaming prefetchers detect accesses to ascending or descending addresses and can prefetch up to 20 lines ahead or behind. Our architecture also has a prefetcher that can detect strides in memory access, as well as a "Next Page Prefetcher" that can load another memory page when detecting memory accesses near the page boundary [26].

11.2 Skewing The Tree

Skewing the tree is done by changing the way we find which character in the alphabet to split on in the construction of each node. The split character is the last character in the alphabet of the left child node and to be able to skew the calculation we calculate it as

$$SplitCharacter = \left\lfloor \frac{alphabetSize - 1}{skew} + alphabetMin \right\rfloor$$

where *alphabetSize* is the size of the alphabet at this node, *alphabetMin* is the first character in the alphabet at this node, and *skew* is the skew parameter which is 2 for a balanced tree and higher for right-skewed trees. E.g. a *skew* value of 4 skews the tree by 75 % to the right so that, in each node, 25 % of the alphabet is put into the left child node and 75 % is put into the right child node. We only use integer values as characters, so the calculated split character is rounded down.

11.3 Controlled Memory Layout

We still want to support dynamic input and alphabet sizes without recompilation, so the nodes must be dynamically allocated on the heap.

The size of a node is known at compile time as it contains fixed-size pointers to the parent node and left and right child nodes, as well as a boolean value to flag it as a

[25] Our architecture is Ivy Bridge, but the optimization manual sections for Sandy Bridge holds true for Ivy Bridge as well, as stated in section 2.2.7 in [23].

[26] See section 2.2.7 of [23]

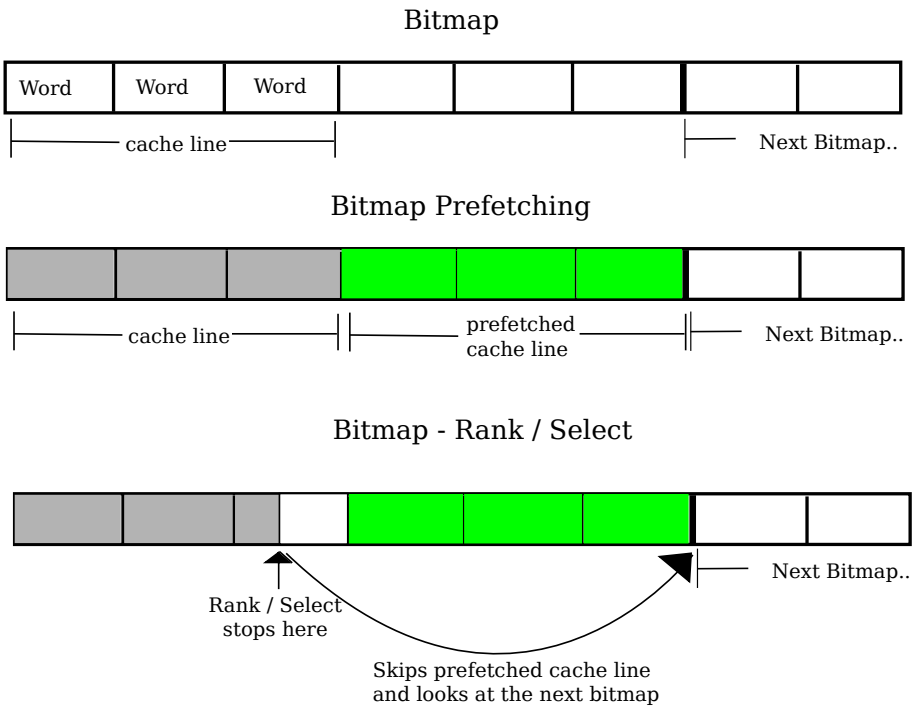


Figure 27: How access patterns in a concatenated bitmap can defeat cache prefetching

leaf node and its bitmap as a vector, which internally stores a pointer to its backing array. As such, the memory for the nodes is allocated by allocating an array and then instantiating the nodes into that array. A reference to a pointer is passed into the array from parent to child nodes during construction, so they know where to allocate their child nodes. The pointer points to the position of the last node in the array, and so before each instantiation of a new node, we increment the pointer so it points to free space, then place the new node there.

The memory layout of the bitmaps are not controlled, because skewing the tree will not help the prefetcher with regards to the bitmaps, except in a few specific cases, because of the way the bitmaps are used and the resulting access patterns. The algorithms for rank and select stop querying each bitmap at some position inside the bitmap and then continue to the next bitmap in the next node. The problem is shown in Figure 27. The drawing assumes the bitmap is stored sequentially and the prefetcher prefetches the next cache line (colored green), but the algorithm stops at some position and skips ahead to the next bitmap. Rank stops when it reaches the position the query was searching up to, given as a parameter. Select stops when it has found the sought number of occurrences in the bitmap. In both of these cases the rest of the bitmap is not used and any such data the prefetcher has fetched would have been in vain. The prefetcher cannot tell from the algorithms access pattern when it will jump ahead to the next bitmap, and every such jump will therefore give rise to a cache miss. This makes it unable to utilize the prefetched data and will try to access memory that is not in the cache yet; a cache miss.

So regardless of where the bitmap that is accessed next is stored, following right after the first or elsewhere in memory, a cache miss will occur.

The exceptions to this are when either the entire bitmap is used for the query, that is, when the rank query is for the entire string, or the bitmap is small enough that the beginning of the next bitmap can fit within the same word. The first case is not a common query in most use cases, and the second case is rare when the input string is much bigger than the alphabet, and would only happen near the leaf nodes. Neither scenario happens often enough to warrant controlling the bitmap memory layouts.

11.4 Experiments

11.4.1 Queries when skewing the Wavelet Tree using uncontrolled and controlled memory layout

In this experiment we want to test whether queries on a skewed tree using controlled memory layout is an improvement in running time and how a change in running time can be explained by the amount of incurred hardware penalties.

Test Setup
The general setup is as described in Section 7.2. The query parameters were chosen as described in Section 7.4. The results can be seen in Figure 28, Figure 29 and Figure 30. We skew the wavelet tree as described in Section 11.2). The block size used for testing the build time and memory usage is 1024, as we found that to be the overall best when weighing both rank and select time and memory usage. The block size used for rank is 64 bit and 2048 bit for select, as that gave the best performance for each, and we wanted to give each the best possibility to perform well when skewed. The less time that is spent in binary rank and binary select, the greater the influence on running time of branch mispredictions and cache misses gets from navigation of nodes in the tree. Skewing the tree can improve running time by reducing these branch mispredictions and cache misses from navigating the tree. Therefore, the less time that is spent in binary rank and binary select, the better opportunity skewing the tree has to improve running time.

Results
Looking at Figure 28a, we can see that it takes more time to build a skewed tree, with increasing time spent as the skew increases. It appears to be a linear increase in running time as skew increases, especially from skew parameter 3 and higher. Looking at Figure 28b we can see that it also takes more memory the more we skew the tree. If we look at Figure 29 and Figure 30 we can see that query times for both rank and select also increase linearly with increasing skew.

We can already conclude that skewing the tree is not worth it in terms of build time, memory usage *or* query times. We still look closer at our measurements and try to explain why it is not worth it.

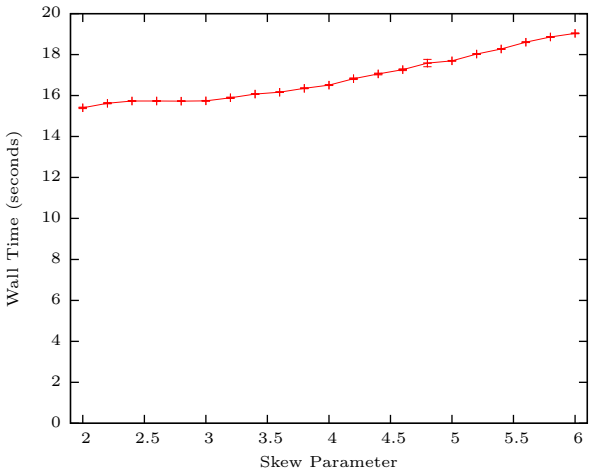
Looking at Figure 29b and Figure 30b we can see the amount of cache misses increases as skew increases for both rank and select queries and for all three levels of cache. If we instead look at level 2 data cache *hits* in Figure 29c and Figure 30c, we can see that they in fact increase as well, and at a higher rate than the cache misses, which is also what the rising cache hit rate signifies. The high increase in level 2 data cache hits is no benefit, however, when level 2 data cache misses still increase, as it is the misses that cause a penalty and no increased amount of cache hits can make up for that penalty.

In Figure 29d and Figure 30d we see an increase in branches executed, which is to be expected, as the depth of the tree increases, more nodes must be traversed using branching code. While more branches are taken overall, we also see in Figure 29e and Figure 30e a reduction in both the amount of branch mispredictions and the branch misprediction rate as the tree is skewed further. The reduction in the raw amount of branch mispredictions leads directly to a reduction in performance penalties incurred, but the increase in total number of conditional branches executed might not be worth it. The amount of branches executed at skew factor 6 compared to skew factor 2 (balanced) increases by 37,149 for rank and 1,365,427 for select, whereas the amount of branch mispredictions are only reduced by 1,377 for rank and 63,116 for select, meaning that there is 21 additional branches per branch misprediction saved for rank and 27 for select. A branch misprediction penalty must then be at least either 21 or 27 cpu cycles before it would be worth it in terms of branch mispredictions, assuming a single branch instruction takes one cpu cycle. Agner Fog has tested the Ivy Bridge architecture and found that the branch misprediction penalty is "15 cycles or more"[27], so it seems the reduction in branch mispredictions is not worth the increase in total branches.

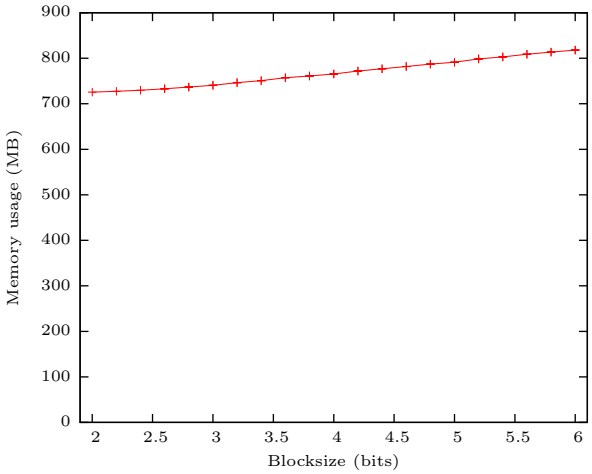
Figure 29f and Figure 30f show the amount of Translation Lookaside Buffer Misses encountered as the tree is skewed. While it has higher variation and both dips and rises the more the tree is skewed, it does still show a general increase at higher skew parameter values.

In the end we can confirm that skewing the tree is no improvement even though the amount of branch mispredictions are reduced. This reduction simply does not make up for the increased amount of cache misses, increased cycles from more executed branches and increased TLB misses. Furthermore, skewing uses more memory.

[27]Section 3.7 in `http://www.agner.org/optimize/microarchitecture.pdf`

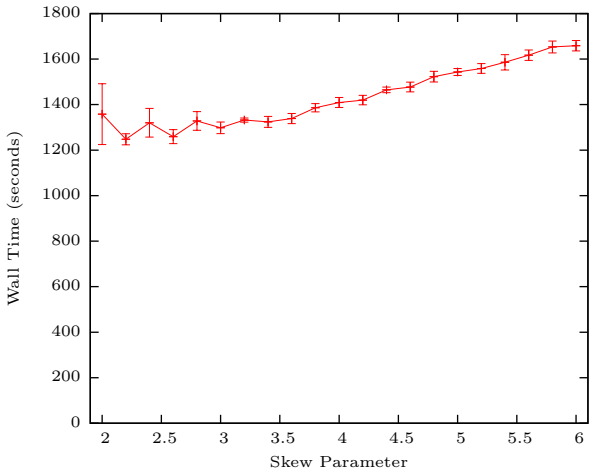


(a) Wall Time

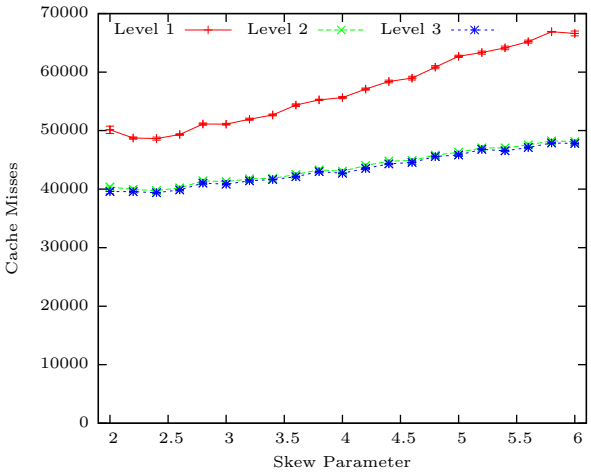


(b) Memory Usage

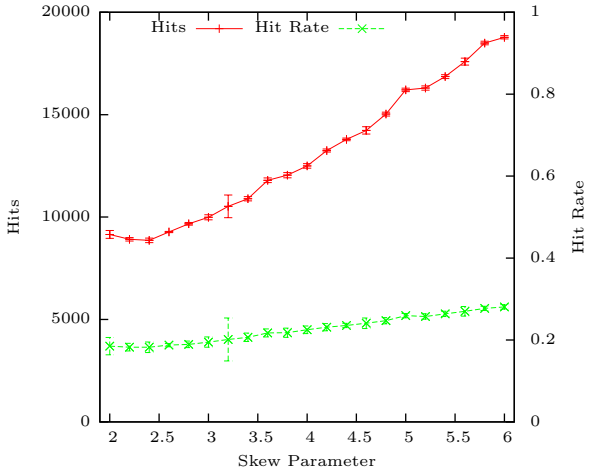
Figure 28: Build Wall Time and Memory Usage of CumulativeSum for various skew.



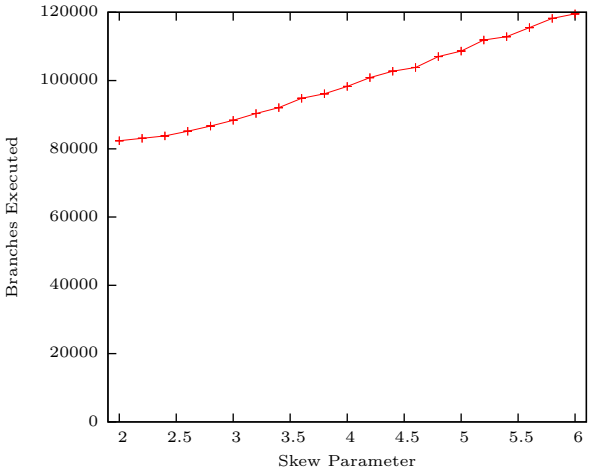
(a) Wall Time



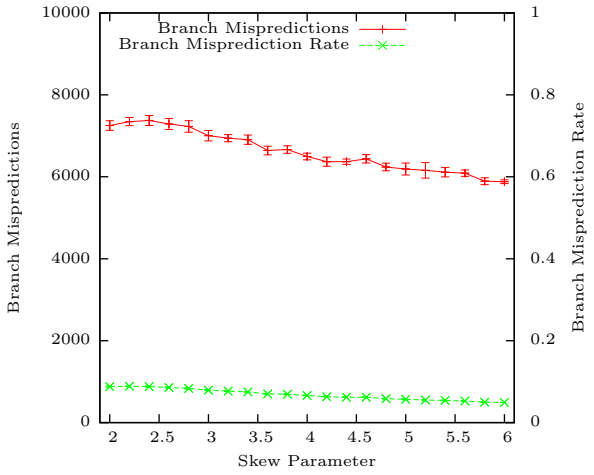
(b) Cache Misses



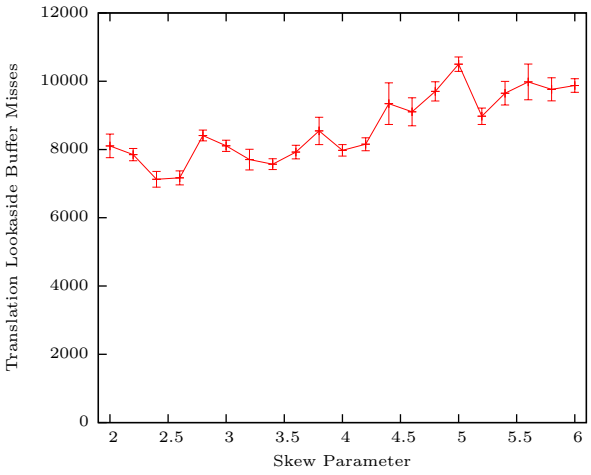
(c) Level 2 Data Cache Hits & Hit Rate



(d) Branches Executed

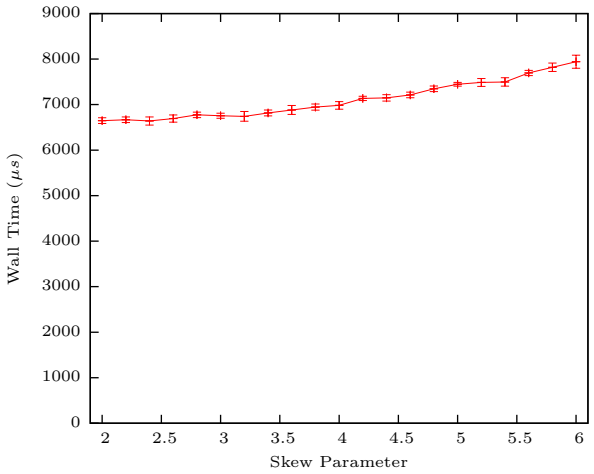


(e) Branch Mispredictions

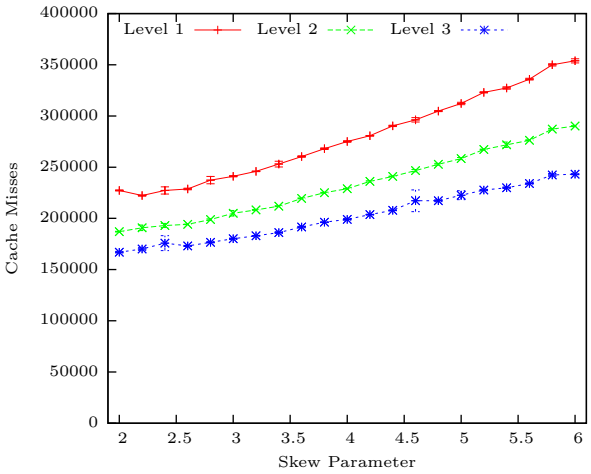


(f) Translation Lookaside Buffer Misses

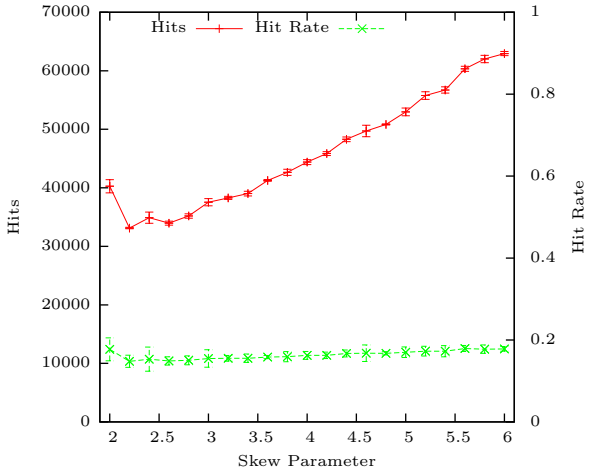
Figure 29: Measurements for Rank Queries on CumulativeSum for various skew.



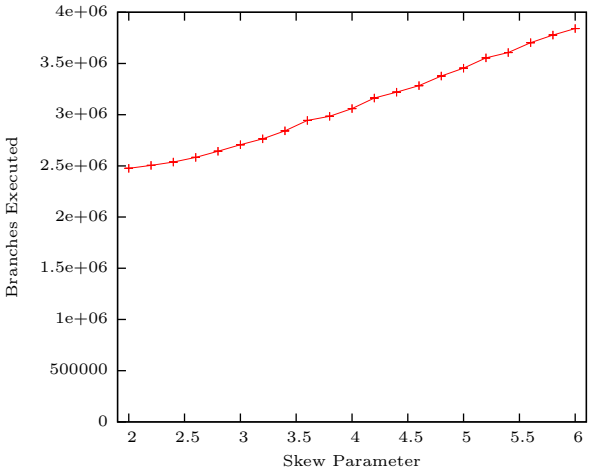
(a) Wall Time



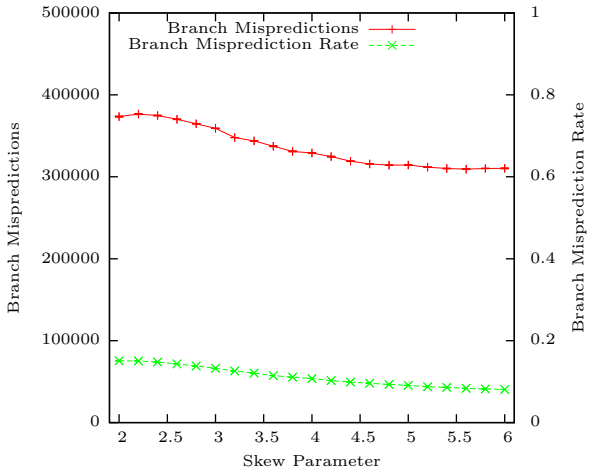
(b) Cache Misses



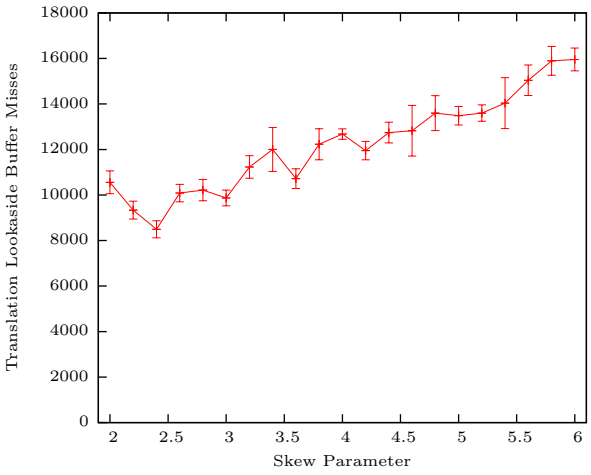
(c) Level 2 Data Cache Hits & Hit Rate



(d) Branches Executed



(e) Branch Mispredictions



(f) Translation Lookaside Buffer Misses

Figure 30: Measurements for Select Queries on CumulativeSum for various skew.

Part IV
Conclusion

12 Conclusion

In this Thesis we have described the wavelet tree, a versatile data structure offering solutions within problem domains such as data compression and information retrieval. We describe in detail how a wavelet tree is constructed and how it is queried in three ways: access, rank, and select.

We have performed a survey on the applications of wavelet trees including efficient data compression and fast information retrieval with more detail on how the wavelet tree is used for some of the applications.

We have described some characteristics of modern CPUs that result in penalties in running time for some cases, such as cache misses (CM), branch mispredictions (BM) and translation lookaside buffer (TLB) misses. We have described how these penalties are incurred and why they result in performance loss.

We have implemented and tested the construction of a wavelet tree, comparing it to the theoretical running time and found the theoretical running time only holds up to a certain alphabet size whereafter an exponentially rising amount of TLB misses reduces performance significantly.

We have implemented rank and select queries and performed a number of modifications, attempting to reduce the amount of hardware penalties they encounter by changing how they are calculated, changing the shape of the tree, changing what is stored and how it is stored.

We have performed a number of experiments to measure and document the change in amount of hardware penalties encountered as well as running times and memory usage to see if our modifications were any improvement. The modifications and the results of the experiments are summed up below.

Using `popcount` CPU instruction to improve binary rank and select query running times within each node of the tree by reducing the amount of CPU cycles needed.
Result: High improvement in running time.

Precompute and store binary rank values in blocks for each bitmap in each node. Use the precomputed values for the most part to reduce the amount of CPU cycles and memory accesses needed.
Result: High improvement in running time.

Concatenate bitmaps and precomputed values to reduce memory usage and possibly improve cache performance.
Result: Small improvement in memory usage, worse running time.

Align bitmaps with memory pages to reduce TLB misses.
Result: Slightly worse running time.

Store cumulative sum of precomputed values instead of raw binary rank values to further reduce CPU cycles and memory accesses needed.

Result: Fastest improvement of running time out of all our optimizations but uses more memory than the others but only 0.5 % more than a wavelet tree with concatenated bitmaps, which is the one that uses the smallest amount of memory.

Replace branching code with arithmetic operations in select queries to reduce number of branches and thereby branch mispredictions.
Result: Worse running time than normal branching select.

Skewing the tree with a controlled memory layout to reduce branch mispredictions.
Result: Branch mispredictions were reduced but it resulted in a worse query running time and increased memory usage.

In general, improvements that reduced the raw amount of computations and memory accesses needed were a big improvement, whereas improvements focusing on reducing a single type of penalty, such as BM or TLB misses, were either barely an improvement or no improvement at all.

13 Future Work

We have many ideas for future work on practical implementation and optimization of wavelet trees.

13.1 Interleaving Bitmap and Precomputed Cumulative Sum Values

Calculating the binary rank using a precomputed cumulative sum value and a bitmap requires a lookup in two separate vectors both of which can introduce a cache miss. If the precomputed cumulative sum values were to be interleaved with the bitmap, so that the precomputed value for a block would lie right next to that block from the bitmap, we expect the second of these cache misses could be avoided.

More precisely, in our implementation two vectors containing different data are stored: one containing the bitmap values and one containing the precomputed values. Instead of this, a new data type could be defined that contained both a block of the bitmap and its precomputed value, and then a single vector containing this data type could then be stored.

We expect this would avoid a cache miss, because the access pattern of a rank query is to access the precomputed value for a block, and then the corresponding block in the bitmap, and if the block from the bitmap is in the same cacheline as the precomputed value, accessing it right after will not lead to a cache miss.

13.2 vEB Memory Layout

We tried a right-side depth-first memory layout in Section 11 when we tried to skew the tree. Without trying to skew the tree, other memory layouts might still be able to improve the performance of the wavelet tree. Brodal et al. [13] tested several memory layouts for their skewed binary search tree and found that the blocked memory layout based on van Emde Boas Trees performed best for all skew values. It could be interesting

to try a van Embe Boas memory layout for a balanced wavelet tree to see if it could improve the query performance.

13.3 d-ary

Alex Bowe [10] has shown that multiary wavelet trees can work in practise. In our implementations we have used a binary wavelet tree which means its height is the base-2 logarithm of the alphabet size. With a d-ary tree the height would be reduced to base d logarithm of the alphabet size. This could improve access, rank, and select query performance significantly as their traversal down or up the tree would be significantly shortened.

A disadvantage of a d-ary wavelet tree is that each bitmap must encode $\log_2(d)$ bits of information for each character in the string, to signify which of the subtrees each character belongs to. This makes using the native `popcount` cpu instruction impossible, perhaps unless some clever bitshifting and `XOR`ring could be applied to avoid manually counting sets of bits. On the other hand, using the stored precomputed values means only few sets of bits would have to be counted and perhaps the benefit from a lower tree will outweigh the loss from not using `popcount`.

13.3.1 SIMD

When constructing or traversing a d-ary wavelet tree, finding which of 4 or more subtrees to either pass a character too or traverse into requires comparing the character with more than just one split character. To improve the performance of this multi-way comparison, SIMD instructions might be employed with success.

13.4 Parallelization

To expand on the potential improvement from using SIMD instructions when constructing and traversing d-ary wavelet trees, some amount of parallelization of the algorithms might improve the performance even further.

13.4.1 On GPU

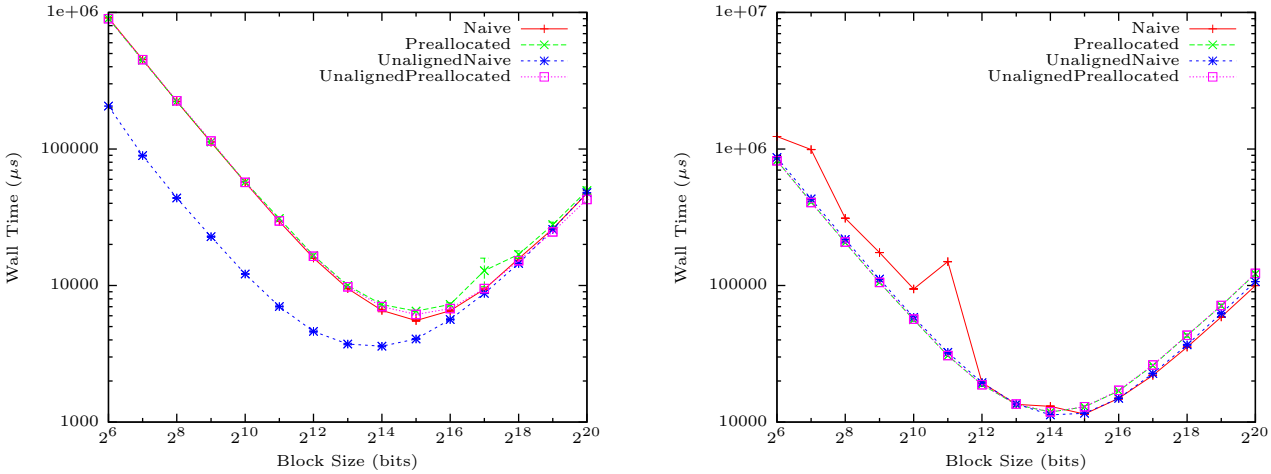
If parallelization proves to be an improvement, implementing them on the GPU e.g. using CUDA could be a massive improvement as modern GPUs have several hundred cores and if well-utilized can surpass the power of a modern CPU.

13.5 RRR structure

The RRR structure allows computation of binary rank in $O(1)$ time. It also implicitly achieves zero-order compression of the data. RRR uses some of the same concepts as we do in our CumulativeSum implementation: Precomputed ranks, cumulative sum of those and concatenation of bitmaps. It could be interesting to measure and analyse the hardware penalties in this structure, and perhaps improve its running time.

Appendices

A Precomputed rank block sizes: larger range



(a) Rank: Wall Time

(b) Select: Wall Time

Figure 31: Running time for Rank and Select queries in Wavelet Trees with Precomputed Rank Values for larger range of varying block sizes. A page size is 2^{15} bits.

Primary Bibliography

[A1] Gonzalo Navarro. Wavelet trees for all. *J. of Discrete Algorithms*, 25:2–20, March 2014. ISSN 1570-8667. doi: 10.1016/j.jda.2013.07.004. [Introduction, Section 4].

[A2] 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, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5. [Section 4.2].

[A3] Cristos Makris. Wavelet trees: a survey. *Computer Science and Information Systems*, 9(2):585–625, 2012. [Introduction, Section 2.1 pages 588-590].

[A4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994. [Introduction].

[A5] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849 – 866, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.010. [Introduction (excluding paragraph C and D)].

[A6] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, volume 3537 of *Lecture Notes in Computer Science*, pages 45–56. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26201-5. doi: 10.1007/11496656_5. [Introduction].

[A7] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In Amihood Amir, Andrew Turpin, and Alistair Moffat, editors, *String Processing and Information Retrieval*, volume 5280 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-89096-6. doi: 10.1007/978-3-540-89097-3_18. [Abstract].

[A8] Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 1–6. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03783-2. doi: 10.1007/978-3-642-03784-9_1. [Section 3].

[A9] Julian Shun. Parallel wavelet tree construction. *CoRR*, abs/1407.8142, 2014. URL `http://arxiv.org/abs/1407.8142`. [Abstract].

[A10] Alex Bowe. Multiary wavelet trees in practice (honours thesis), 2010. URL `https://github.com/alexbowe/wavelet-paper/raw/thesis/thesis.pdf`. [Abstract].

[A11] Andrew S. Tanenbaum. *Structured Computer Organization (5th Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. ISBN 0131485210. [Sections 4.5.1, 4.5.2].

[A12] L. Bic and A.C. Shaw. *Operating Systems Principles.* An Alan R. Apt Book. Prentice Hall, 2003. ISBN 9780130266118. [Section 8.2.1 (Paging, Page Tables), Section 8.2.2 (Segmentation, Paging with Segmentation), Section 8.2.5].

[A13] Gerth Stølting Brodal and Gabriel Moruz. Skewed binary search trees. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 708–719. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-38875-3. doi: 10.1007/11841036_63. [Introduction].

[A14] Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Rolf Fagerberg Riko Jacob. Cache oblivious search trees via binary trees of small height. In *In Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 39–48, 2002. [Abstract].

Secondary Bibliography (not curriculum)

[B15] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *CoRR*, abs/0705.0552, 2007. URL `http://arxiv.org/abs/0705.0552`.

[B16] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952. ISSN 0096-8390. doi: 10.1109/JRPROC.1952.273898. [Summary].

[B17] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398, 2000. doi: 10.1109/SFCS.2000.892127. [Section 3 (excluding subsections)].

[B18] Vreda Pieterse, Derrick G. Kourie, Loek Cleophas, and Bruce W. Watson. Performance of c++ bit-vector implementations. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 242–250, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-950-3. doi: 10.1145/1899503.1899530.

[B19] Steven T. Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions (abstract). *Psychonomic Bulletin & Review*, 21:1112–1130, 2014. ISSN 1069-9384. doi: 10.3758/s13423-014-0585-6. [Introduction].

[B20] C. Browne, B. Culligan, and J. Phillips. The new general service list, 2013. URL `http://newgeneralservicelist.org`.

[B21] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05) (Greece*, pages 27–38, 2005.

[B22] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 137–142, New York, NY, USA, 1972. ACM. doi: 10.1145/800152.804906.

[B23] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, September 2014. URL `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`. [Section 2.2.7].