Master's thesis

Master's Programme in Computer Science

Practical Aspects of Implementing a Suffix Array-based Lempel-Ziv Data Compressor

Aki Utoslahti

May 15, 2022

Faculty of Science

University of Helsinki

Supervisor(s)

Assoc. Prof. Simon J. Puglisi

Examiner(s)

Assoc. Prof. Simon J. Puglisi
Dr. Juha Kärkkäinen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Aki Utoslahti

Työn nimi — Arbetets titel — Title

Practical Aspects of Implementing a Suffix Array-based Lempel-Ziv Data Compressor

Ohjaajat — Handledare — Supervisors

Assoc. Prof. Simon J. Puglisi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 15, 2022	91 pages, 25 appendix pages

Tiivistelmä — Referat — Abstract

Lempel-Ziv factorization of a string is a fundamental tool that is used by myriad data compressors. Despite its optimality regarding the number of produced factors, it is rarely used without modification, for reasons of its computational cost. In recent years, Lempel-Ziv factorization has been a busy research subject, and we have witnessed the state-of-the-art being completely changed. In this thesis, I explore the properties of the latest suffix array-based Lempel-Ziv factorization algorithms, while I experiment with turning them into an efficient general-purpose data compressor.

The setting of this thesis is purely exploratory, guided by reliable and repeatable benchmarking. I explore all aspects of the suffix array-based Lempel-Ziv data compressor. I describe how the chosen factorization method affects the development of encoding and other components of a functional data compressor. I show how the chosen factorization technique, together with capabilities of modern hardware, allows determining the length of the longest common prefix of two strings over 80% faster compared to the baseline approach. I also present a novel approach to optimizing the encoding cost of the Lempel-Ziv factorization of a string, i.e., bit-optimality, using a dynamic programming approach to the Single-Source Shortest Path problem.

I observed that, in its current state, the process of suffix array construction is a major computational bottleneck in suffix array-based Lempel-Ziv factorization. Additionally, using a suffix array to produce a Lempel-Ziv factorization leads to optimality regarding the number of factors, which does not necessarily correspond to bit-optimality. Finally, a comparison with common third-party data compressors revealed that relying exclusively on Lempel-Ziv factorization prevents reaching the highest compression efficiency. For these reasons, I conclude that current suffix array-based Lempel-Ziv factorization is unsuitable for general-purpose data compression.

ACM Computing Classification System (CCS)
Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Data compression

Avainsanat — Nyckelord — Keywords

Data compression, Lempel-Ziv factorization, Suffix array, Integer coding, String comparison

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction

For more than 40 years, the Lempel-Ziv factorization [14] has been a cornerstone of data compression and general stringology. Its first application to data compression dates back to the LZ77 data compression algorithm [30], and since then it has played an important role within modern data compressors, for example, in `gzip`, `WinZip`, `7-Zip`, and `lz4`. It also acts as a general measure of compressibility, as the size of the Lempel-Ziv factorization of a string has been shown to be a lower bound for the size of the smallest context-free grammar that represents it [2]. Other recent applications include finding maximal repetitions in a string [12], detecting periodicities in strings [4], and compressed full-text indexes intended for searching highly repetitive collections [17; 16]. The common aspect between all these applications is that the calculation of the Lempel-Ziv factorization is, in practice, a bottleneck in their time and memory consumption [10]. Because of that, Lempel-Ziv factorization has remained a busy research target.

During the last two decades, the state-of-the-art of Lempel-Ziv factorization algorithms has changed completely. The development dates back to 2008, when Crochemore and Ilie showed how the *LPF* (Longest Previous Factor) array of a string could be computed in linear time, using only the corresponding suffix array [3]. The *LPF* array leads directly to linear-time Lempel-Ziv factorization. Before Crochemore and Ilie's algorithm, suffix array-based Lempel-Ziv factorization had relied on using additional information, usually the *LCP* (Longest Common Prefix) array. Although the *LCP* array can also be computed in linear time, it is nevertheless a costly extra step that adds to the total computation time and to the space consumption. This led to further developments, with Ohlebusch and Gog showing in 2011 how the technique of Crochemore and Ilie could be performed faster and using less space [19]. Next, Goto and Bannai [7], as well as Kempa and Puglisi [11], showed simultaneously and independently similar ideas on how the computation of the full *LPF* array could be avoided. The algorithms in [7] were based on the observation that the *LPF* array contains more information than is needed for Lempel-Ziv factorization, as only a subset of the *LPF* array entries are used. These algorithms had smaller memory footprint than their predecessors, and they were faster in practice. Finally, later in 2013, Kärkkäinen et al. [10] showed how the algorithms of Goto and Bannai in [7] could be combined and reorganized to further reduce memory usage and running times. Since then, the algorithms of Kärkkäinen et al. have remained the fastest and the most succinct.

Even if Lempel-Ziv factorization is central to many modern data compressors, it is only rarely used without modification or without being accompanied by heuristics because of its computational cost. Although these modifications and heuristics improve the running time, they usually come with a cost to compression performance. For example, the compression algorithm used by `gzip` and `WinZip` restricts the size of the search window used in factorization, while `lz4` performs only a best-effort factorization using a hash table. My main hypothesis in this thesis is that the current state-of-the-art Lempel-Ziv factorization algorithm could be turned into a general-purpose data compressor that would compress arbitrary data in a fast and efficient manner.

The main objective of this thesis is, therefore, to document the process of designing and implementing a prototype of a general-purpose data compressor, using the latest Lempel-Ziv factorization algorithms based on suffix arrays. This includes mapping out the advantages and disadvantages of such an approach, as well as determining its profitability. There are also three secondary objectives. The first objective is to determine what kind of influence the chosen factorization method has on the encoding. The second objective is to explore and map the possibilities to further improve and optimize the chosen factorization method. The last objective is to document the general practical aspects of designing and implementing a general-purpose data compressor. The research setting of this thesis is experimental and exploratory, and as such, I use rapid prototyping, guided by exact and repeatable benchmarks, in a trial-and-error approach to develop the prototype data compressor.

To my knowledge, the algorithms of [10] have not been tested before for general data compression. Additionally, there seem to be no data compressors using Lempel-Ziv factorization based on a suffix array. In this thesis, I also explore the optimality of the Lempel-Ziv factorization with respect to the size of the encoding, that is, bit-optimality as opposed to the number of factors in the factorization. Research on the bit-optimality of Lempel-Ziv factorization is extremely sparse, and the most notable works include those of Horspool [8] and Ferragina et al. [5]. Therefore, this thesis potentially provides novel information on topics that have received little or no attention.

This thesis documents the design and implementation of a suffix array-based Lempel-Ziv data compressor from the ground up and is structured as follows. First, Chapter 2 introduces the central background concepts used in the development of the building blocks of the prototype data compressors. Next, Chapter 3 describes in detail the development steps of these building blocks. In Chapter 4, I define the final prototype data compressor

designs built using those blocks, describe how they were benchmarked, and provide a comparison between them. Finally, in Chapter 5, I compare the prototype data compressor designs with well-known third-party data compressors, define points of interest for future research, and provide my final conclusions.

2 Background

In this chapter, I introduce the background concepts that are central to the methods discussed in Chapter 3. I start by defining common terms and notation that are used throughout this thesis. I then introduce the suffix array data structure and the Lempel-Ziv factorization, before describing how a state-of-the-art Lempel-Ziv factorization algorithm can be designed using the suffix array. Finally, I describe three integer coding schemes that were utilized in the prorotype data compressor developed during the research phase of this thesis.

2.1 Common terminology

This section describes the common terminology and notation used in this thesis. Those that are related to *texts/strings* are mostly borrowed from Kärkkäinen et al. [10] and Crochemore and Ilie [3].

Byte units. Two different orders of magnitude for byte units are used in this thesis, decimal (SI) and binary (IEC). The decimal prefixes are powers of 10^3 and named as *kilo (k)*, *mega (M)* and *giga (G)*. The corresponding binary prefixes are powers of 2^{10} and named as *kibi (ki)*, *mebi (Mi)* and *gibi (Gi)*. For example, 1 megabyte (1 MB) refers to $1,000,000$ bytes, while 1 mebibyte (1 MiB) refers to $1,048,576$ bytes.

Compression ratio. The compression ratio is the relative reduction in space usage achieved with data compression, i.e., the level of compression.

$$\text{compression ratio} = \frac{\text{uncompressed size}}{\text{compressed size}}$$

Text. A text $T = T[1 \dots n] = T[1]T[2] \dots T[n]$ is a concatenation of $|T| = n$ symbols drawn from an *alphabet* Σ of size $|\Sigma| = \sigma$. The alphabet model being addressed in this thesis is the *byte alphabet* ($\Sigma = \{0, \dots, 255\}$), which is a special case of an *integer alphabet* ($\Sigma = \{0, \dots, \sigma - 1\}$). An integer alphabet is ordered and therefore it supports symbol comparison with operators $<$ (less than) and $=$ (equal to). Symbols of an integer alphabet can also be used in arithmetic and as array indices. In this thesis, I use the terms *text* and *string* synonymously.

Substring. A substring of text $T[1 \dots n]$ that starts at position i and ends at position j, such that $i, j \in \{1, \dots, n\}$ and $i \leq j$, is denoted by $T[i \dots j] = T[i]T[i+1] \dots T[j]$. In this thesis, I use the terms *substring* and *factor* synonymously.

Prefix. A prefix of text $T[1 \dots n]$ is a substring that starts at position 1 and ends at position $i \in \{1, \dots, n\}$, which is denoted by $T[1 \dots i] = T[1]T[2] \dots T[i]$. In this thesis, I use "prefix i" to refer to prefix $T[1 \dots i]$.

Suffix. A suffix of text $T[1 \dots n]$ is a substring that starts at position $i \in \{1, \dots, n\}$ and ends at position n, which is denoted by $T[i \dots n] = T[i]T[i+1] \dots T[n]$. In this thesis, I use "suffix i" to refer to suffix $T[i \dots n]$.

Longest common prefix. The length of the longest common prefix of suffixes i and j of text $T[1 \dots n]$, denoted by $lcp(i, j)$, is the maximum $l \leq \min(|T[i \dots n]|, |T[j \dots n]|)$ such that $T[i \dots i+l-1] = T[j \dots j+l-1]$. For compatibility with algorithms in the following sections, I define $lcp(0, i) = lcp(i, 0) = 0$ for $i \in \{1, \dots, n\}$.

Lexicographical order. Let i and j be suffixes of text $T[1 \dots n]$, and let $lcp(i, j) = l$. Suffix i is lexicographically smaller than or equal to suffix j, denoted by $T[i \dots n] \leq T[j \dots n]$, if and only if either

- $|T[i \dots n]| = l$, or
- $|T[i \dots n]| > l$, $|T[j \dots n]| > l$ and $T[i + l] \leq T[j + l]$.

2.2 Suffix array

The suffix array is a data structure introduced by Manber and Myers in 1990, as a space efficient alternative to suffix trees for on-line string searches [15]. Since then, suffix arrays have been a vibrant area of research and they have been widely used in full-text indices, data compression, and stringology.

Suffix array. The suffix array SA of text $T[1 \dots n]$ is a lexicographically ordered array of all suffixes of T. Importantly, the suffix array is not an array of texts, but instead an array of integers. This is possible as each suffix can be uniquely determined by its starting position. Therefore, suffix array SA is a permutation of integers $[1 \dots n]$ such that

$$T[SA[1] \dots n] \leq T[SA[2] \dots n] \leq \dots \leq T[SA[n] \dots n].$$

For compatibility with the algorithms in following sections, I define $SA[0] = SA[n+1] = 0$.

Inverse suffix array. The inverse suffix array ISA is the inverse permutation of the suffix array. Let $T[1 \dots n]$ be a text. For each suffix $i \in \{1, \dots, n\}$, $ISA[i]$ contains the position of suffix i in the suffix array. Formally, for $i, j \in \{1, \dots, n\}$, $ISA[i] = j$ if and only if $SA[j] = i$.

The suffix array and the inverse suffix array for the text $T =$ "banana" are illustrated together in Table 2.1.

Table 2.1: Suffix array for text $T =$ "banana".

i	**T[i . . . n]**	**ISA[i]**	**SA[i]**	**T[SA[i] . . . n]**
1	"banana"	4	6	"a"
2	"anana"	3	4	"ana"
3	"nana"	6	2	"anana"
4	"ana"	2	1	"banana"
5	"na"	5	5	"na"
6	"a"	1	3	"nana"

Suffix array construction for text $T[1 \dots n]$ of integer alphabet is possible using $O(n)$ time and $O(n \log n)$ bits of space [18]. A complete suffix array uses $n \log n$ bits of space for any alphabet.

2.3 Lempel-Ziv factorization

Dictionary-based data compression algorithms replace parts of a text with references to a dictionary. Actual compression is achieved when the references use less space than the parts of text that they replace. In 1977, Lempel and Ziv introduced the LZ77 data compression algorithm, which utilizes the preceding part of the text as a dictionary [30]. Even though the original LZ77 data compression algorithm has been bested by numerous derivatives, its core principle, the Lempel-Ziv factorization [14], still remains a fundamental tool for data compression.

Longest previous factor. The longest previous factor for position i of text T is the longest substring that occurs both at position i and to the left of it in T.

Conceptually, the Lempel-Ziv factorization is a greedy left-to-right factorization of a text into its longest previous factors, encoded as tokens. In this thesis, I use "LZ77 factor-

ization" to refer to Lempel-Ziv factorization that is encoded using LZ77 token format. I represent the LZ77 tokens as $(o/s, l)$ pairs, where

- o/s: offset/symbol, represents offset, if a previous occurrence exists, or a symbol literal otherwise.
- l: length, represents the length of the occurrence, i.e., $l > 0$ if previous occurrence exists, or $l = 0$ otherwise.

For example, the LZ77 factorization algorithm would partition text $T =$ "bananabandana" into factors `b|a|n|ana|ban|d|ana` and encode them using the above token format as:

$$(\overset{\text{'b'}}{\text{'b'}}, 0), (\overset{\text{'a'}}{\text{'a'}}, 0), (\overset{\text{'n'}}{\text{'n'}}, 0), (\overset{\text{"ana"}}{2}, 3), (\overset{\text{"ban"}}{6}, 3), (\overset{\text{'d'}}{\text{'d'}}, 0), (\overset{\text{"ana"}}{7}, 3).$$

Algorithm 1 shows a naïve implementation of the LZ77 factorization algorithm. For each factor starting position p of text $T[1 \dots n]$, the algorithm finds the longest previous factor by computing the longest common prefix between suffix p and all $p-1$ preceding suffixes. The time complexity of the naïve LZ77 factorization algorithm is $O(n^2)$, as the total number of symbol comparisons is bounded by the total length of the longest previous factors, i.e., the length of the text n [7]. The algorithm to reconstruct text T, given its LZ77 factorization, is shown in Algorithm 2. For each factor, the algorithm first checks if it is a symbol literal or a reference to preceding, already reconstructed, text. If the factor is a symbol literal, the algorithm simply appends the symbol to the already reconstructed text. Otherwise, it copies a substring of l symbols, from a position that is o symbols to the left of the current position, to the end of the already reconstructed text. The algorithm runs in $O(n)$ time as the total amount of work is bounded by the length of the text. Note that this same reconstruction algorithm is also utilized with the advanced factorization algorithms introduced in the next section. Both factorization and reconstruction algorithm use only $O(1)$ extra space.

The main disadvantage with the previously introduced LZ77 token format is the high likelihood of increased space usage due to encoding all factors using the same format. In particular, a pair of integers for each single repeated symbol, or symbol that has not occurred before, is wasteful. In 1982, Storer and Szymanski introduced LZSS (Lempel-Ziv-Storer-Szymanski), a derivative of LZ77, which addresses this problem (among others) [26]. LZSS strives to improve LZ77 in multiple ways, but in the context of this thesis, we are only interested in two of them. First, instead of using exclusively pairs to represent the

Algorithm 1 Naïve LZ77 factorization of text $T[1 \dots n]$.

```
1:  p ← 1
2:  while p ≤ n do
3:      length ← 0
4:      offset ← 0
5:      for i ← 1, ..., p − 1 do
6:          l ← lcp(i, p)
7:          if l > length then
8:              offset ← p − i
9:              length ← l
10:     if length > 0 then
11:         OUTPUT-FACTOR((offset, length))
12:     else
13:         OUTPUT-FACTOR((T[p], 0))
14:     p ← p + max(1, length)
```

Algorithm 2 Reconstruction of text T, given its LZ77 factorization $((o_1, l_1), \dots, (o_z, l_z))$.

```
1:  p ← 1
2:  for i ← 1, ..., z do
3:      if l_i = 0 then
4:          T[p] ← o_i
5:      else
6:          for j ← 0, ..., l_i − 1 do
7:              T[p + j] ← T[p − o + j]
8:      p ← p + max(1, l_i)
```

factorization, LZSS uses one-bit flags to indicate whether a factor is an offset/length pair or a symbol literal. Second, Storer and Szymanski had an idea to discard offset/length pairs which use more space than the original text they replace.[1] By using unset bits to represent literals and set bits to represent factors, we obtain following LZSS factorization for the example text:

$$\overset{flags}{0001101}, \overset{\text{'b'}}{\text{'b'}}, \overset{\text{'a'}}{\text{'a'}}, \overset{\text{'n'}}{\text{'n'}}, \overset{\text{"ana"}}{(2,3)}, \overset{\text{"ban"}}{(6,3)}, \overset{\text{'d'}}{\text{'d'}}, \overset{\text{"ana"}}{(7,3)}.$$

In this thesis, I make a distinction between (normal) factors that are defined by an offset/length pair, and factors that are symbol literals. From this point forward, I use "factor" to refer exclusively to former, and "literal" to refer to latter.

2.4 Suffix array-based Lempel-Ziv factorization

In the previous section, I showed a naïve LZ77 factorization algorithm that runs in $O(n^2)$ time using $O(1)$ extra space. Even though the memory usage is impressive, the time usage is too great for the algorithm to have much practical use. In this section, I first provide a bottom-up description of the BGS algorithm by Goto and Bannai [7], which is heavily influenced by earlier research of Crochemore and Ilie [3], and Ohlebusch and Gog [19]. BGS is a suffix array-based Lempel-Ziv factorization algorithm that runs in $O(n)$ time using $(4n + s)\log n$ bits of extra space, where $s \leq n$ is the maximum size of the stack used by the algorithm. Afterwards, I describe how KKP3, the current state-of-the-art LZ77 factorization algorithm by Kärkkäinen et al. [10], optimizes the BGS algorithm in order to achieve faster running times, while utilizing only $3n \log n$ bits of extra space.

2.4.1 BGS

The first step in the bottom-up description of BGS algorithm is to define LZ77 factorization using LPF and $PrevOcc$ arrays [3; 7; 10].

LPF and PrevOcc arrays. Let $T[1 \ldots n]$ be a text. For each $i \in \{1, \ldots, n\}$, $LPF[i]$ contains the length of the longest previous factor for suffix i, and $PrevOcc[i]$ contains the starting position of it. If $T[i]$ is the leftmost occurrence of symbol in T, and as a result $PrevOcc[i]$ is not defined, we set $PrevOcc[i] = 0$. If there are multiple candidates for $PrevOcc[i]$, any

[1] This idea is re-explored later in Section 3.5.

of them can be chosen.

$$LPF[i] = \max(lcp(i,j) \mid 1 \leq j < i)$$

$$PrevOcc[i] = \begin{cases} 0 & \text{if } LPF[i] = 0 \\ j & \text{otherwise} \end{cases}$$

LZ77 factorization. Let the factorization be computed up to the position i. If $LPF[i] > 0$, then the next factor is $(i - PrevOcc[i], LPF[i])$. Otherwise, the next factor is $(T[i], 0)$. Table 2.2 contains the LPF and $PrevOcc$ arrays for text $T =$ "bananabandana", while Algorithm 3 shows an algorithm that, given corresponding LPF and $PrevOcc$ arrays, produces LZ77 factorization of text. Note that the actual factorization part is now linear to the number of factors $z = O(n/\log_\sigma n)$ [9], instead of the length of the text n.

Table 2.2: LPF and $PrevOcc$ arrays for text $T =$ "bananabandana"

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$T[i]$	b	a	n	a	n	a	b	a	n	d	a	n	a
$LPF[i]$	0	0	0	3	2	1	3	2	1	0	3	2	1
$PrevOcc[i]$	0	0	0	2	3	4	1	4	5	0	4	5	11

Algorithm 3 LZ77 factorization of text $T[1 \ldots n]$, given its LPF and $PrevOcc$ arrays.

```
1: p ← 1
2: while p ≤ n do
3:     if LPF[p] > 0 then
4:         OUTPUT-FACTOR((p − PrevOcc[p], LPF[p]))
5:     else
6:         OUTPUT-FACTOR((T[p], 0))
7:     p ← max(1, LPF[p])
```

Crochemore and Ilie showed in [3] how the LPF and $PrevOcc$ arrays can be obtained using the suffix array in $O(n)$ time. From the lexicographic ordering of the suffix array, it follows that for any suffix i, the suffixes that appear closer to it in the suffix array, will have longer longest common prefixes with it. Therefore, to find out the longest previous factor for suffix i, it suffices to consider two suffixes, the nearest lexicographic predecessor and successor that have values smaller than i.

PSV_{lex} and NSV_{lex} arrays. Let $SA[1 \dots n]$ be a suffix array of text $T[1 \dots n]$. For each position $i \in \{1, \dots, n\}$, $PSV_{lex}[i]$ contains the position of the *previous smaller value (psv)* in SA, compared to $SA[i]$. If such value does not exist, we set $PSV_{lex}[i] = 0$. $NSV_{lex}[i]$ is similar with the exception that it contains the position of the *next smaller value (nsv)* instead.

$$PSV_{lex}[i] = \max(\{0\} \cup \{1 \leq j < i \mid SA[j] < SA[i]\})$$
$$NSV_{lex}[i] = \min(\{0\} \cup \{i < j \leq n \mid SA[j] < SA[i]\})$$

For each position $i \in \{1, \dots, n\}$ of text $T[1 \dots n]$, the corresponding previous and next smaller values, psv_i and nsv_i, can be obtained using SA, ISA, PSV_{lex} and NSV_{lex} of T.

$$psv_i = SA[PSV_{lex}[ISA[i]]]$$
$$nsv_i = SA[NSV_{lex}[ISA[i]]]$$

Next we redefine $LPF[i]$ and $PrevOcc[i]$ using psv_i and nsv_i.

$$l_{psv_i} = lcp(psv_i, i)$$
$$l_{nsv_i} = lcp(nsv_i, i)$$
$$LPF[i] = \max(l_{psv_i}, l_{nsv_i})$$
$$PrevOcc[i] = \begin{cases} psv_i & \text{if } l_{psv_i} \geq l_{nsv_i} \\ nsv_i & \text{otherwise} \end{cases}$$

Note that the actual LPF and $PrevOcc$ arrays are no longer needed, as it is possible to compute the factorization directly from SA, ISA, PSV_{lex} and NSV_{lex} arrays in a lazy manner. The arrays are demonstrated in Table 2.3, while the corresponding factorization algorithm is illustrated in Algorithm 4. Following similar reasoning as with the naïve LZ77 factorization algorithm, the time complexity of this algorithm is $O(n)$, as for each factor we only consider a constant number of factor candidates.

Since we know (from Section 2.2) that the SA and ISA can be computed in $O(n)$ time, it remains to show how the PSV_{lex} and NSV_{lex} arrays can be computed in $O(n)$ time. Fortunately, it is possible to compute them with a simple linear scan over the suffix array, as shown in Algorithm 5 (adapted from [7]). The algorithm uses a stack as a helper data structure, which during the execution of the algorithm has a maximum size of $s \leq n$ bytes. In practice, s is almost always significantly smaller than n, but in the worst case, when all n elements are pushed to the stack before popping any element, the size is equal to n.

Table 2.3: SA, ISA, PSV_{lex} and NSV_{lex} arrays for text $T =$ "bananabandana".

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$T[i]$	b	a	n	a	n	a	b	a	n	d	a	n	a
$SA[i]$	13	6	11	4	2	8	1	7	10	12	5	3	9
$ISA[i]$	7	5	12	4	11	2	8	6	13	9	3	10	1
$PSV_{lex}[i]$	0	0	2	0	0	5	0	7	8	9	7	7	12
$NSV_{lex}[i]$	2	4	4	5	7	7	0	11	11	11	12	0	0

Algorithm 4 BGS (post-processing): LZ77 factorization of text $T[1 \dots n]$, given its SA, ISA, PSV_{lex} and NSV_{lex} arrays.

1: $p \leftarrow 1$
2: **while** $p \leq n$ **do**
3: $\quad LPF \leftarrow 0$
4: $\quad PrevOcc \leftarrow 0$
5: $\quad psv \leftarrow SA[PSV_{lex}[ISA[p]]]$
6: $\quad nsv \leftarrow SA[NSV_{lex}[ISA[p]]]$
7: $\quad l_{psv} \leftarrow lcp(psv, p)$
8: $\quad l_{nsv} \leftarrow lcp(nsv, p)$
9: $\quad$ **if** $l_{psv} \geq l_{nsv}$ **then**
10: $\quad\quad LPF \leftarrow l_{psv}$
11: $\quad\quad PrevOcc \leftarrow psv$
12: $\quad$ **else**
13: $\quad\quad LPF \leftarrow l_{nsv}$
14: $\quad\quad PrevOcc \leftarrow nsv$
15: $\quad$ **if** $LPF > 0$ **then**
16: $\quad\quad$ OUTPUT-FACTOR$((p - PrevOcc, LPF))$
17: $\quad$ **else**
18: $\quad\quad$ OUTPUT-FACTOR$((T[p], 0))$
19: $\quad p \leftarrow p + \max(1, LPF)$

Therefore, the BGS algorithm can produce the LZ77 factorization of a text in $O(n)$ time using $(4n + s) \log n$ bits of space, as each of the four auxiliary arrays uses $n \log n$ bits of space.

Algorithm 5 BGS (preprocessing): computation of PSV_{lex} and NSV_{lex} arrays of text $T[1 \dots n]$, given its SA.

```
1:  Let S be an empty stack
2:  for i ← 1, ..., n do
3:      while not S.empty() and SA[i] < SA[S.peek()] do
4:          NSV_lex[S.pop()] ← i
5:      if S.empty() then
6:          PSV_lex[i] ← 0
7:      else
8:          PSV_lex[i] ← S.peek()
9:      S.push(i)
10: while not S.empty() do
11:     NSV_lex[S.pop()] ← 0
```

2.4.2 KKP3

The KKP3 algorithm can be essentially seen as an optimized version of the BGS algorithm, and therefore it is simplest to explain it in terms of individual optimizations. In total, there are four different optimizations, which are all introduced in [10].

The first optimization reduces the number of symbol comparisons during LZ77 factorization stage. It is based on observation that $lcp(psv_i, nsv_i) = \min(lcp(psv_i, i), lcp(nsv_i, i))$, which allows reducing the number of symbol comparisons by $lcp(psv_i, nsv_i)$ for each factor i. The optimization can be observed by comparing lines 15–22 of Algorithm 6 to lines 7–14 of Algorithm 4.

The second optimization reduces memory usage and the number of memory accesses during the factorization stage. The KKP3 algorithm uses PSV_{text} and NSV_{text} arrays, which contain the actual psv and nsv values in text order, instead of PSV_{lex} and NSV_{lex} arrays, which contain the positions of psv and nsv values in the suffix array, in lexicographic order. This means that the suffix array and inverse suffix array are not needed anymore in the factorization stage of the algorithm. Therefore, the inverse suffix array is no

longer needed at all, which results in a reduction of $n \log n$ bits in memory usage. The optimization also reduces the number of memory accesses during the factorization stage. PSV_{text} and NSV_{text} arrays are defined in relation to PSV_{lex} and NSV_{lex} arrays such that

$$PSV_{text}[SA[i]] = SA[PSV_{lex}[i]]$$
$$NSV_{text}[SA[i]] = SA[NSV_{lex}[i]].$$

The arrays are shown in Table 2.4, while the optimization can be observed by comparing lines 13–14 of Algorithm 6 to lines 5–6 of Algorithm 4.

Table 2.4: SA, PSV_{text} and NSV_{text} arrays for text $T =$ "bananabandana".

i	1	2	3	4	5	6	7	8	9	10	11	12	13
$T[i]$	b	a	n	a	n	a	b	a	n	d	a	n	a
$SA[i]$	13	6	11	4	2	8	1	7	10	12	5	3	9
$PSV_{text}[i]$	0	0	1	0	1	0	1	2	3	7	6	10	0
$NSV_{text}[i]$	0	1	0	2	3	4	5	1	0	5	4	5	6

The third and fourth optimization further reduce the memory usage, and improve the CPU cache utilization in the preprocessing stage of the algorithm. The third optimization is based on observation by Kärkkäinen et al., that the size of the stack is never larger than the already processed part of the suffix array. Therefore, the optimization removes the external stack and relocates its contents to the already processed part of the suffix array. Note that this optimization should not be used in case the contents of the suffix array should be preserved for later use.

Goto and Bannai observed in [7] that memory accesses to the PSV and NSV arrays always occur at indices that are same, or close to each other. Therefore, they improved the locality of memory access by interleaving the PSV and NSV arrays to a single PSV/NSV array of $2n \log n$ bits, such that

$$PSV[i] = PSV/NSV[2i]$$
$$NSV[i] = PSV/NSV[2i+1].^{1}$$

The KKP3 algorithm does similarly and stores PSV_{text} and NSV_{text} arrays interleaved. However, the KKP3 algorithm optimizes the locality of memory access even further by computing the PSV_{text} values when popping from the stack instead of when pushing to the

[1] The ordering (*lex/text*) has no effect.

stack, so that each $PSV_{text}[i]$ and $NSV_{text}[i]$ are computed and written at the same time. These last two optimizations are illustrated together on lines 1–8 of Algorithm 6, which can be compared to PSV_{lex} and NSV_{lex} construction algorithm shown in Algorithm 5.

The above optimizations together result in a total extra memory usage of $3n \log n$ bits, and faster running time in practice, even if the asymptotic time complexity remains unchanged. It should be noted that Kärkkäinen et al. were further able to reduce the space utilization in [10], but those more succinct algorithms were not able to match the running times of KKP3 in general.

Algorithm 6 KKP3: LZ77 factorization of text $T[1 \dots n]$, given its SA.

```
1:  top ← 0                                                    ▷ Stack top pointer
2:  for i ← 1, ..., n + 1 do
3:      while SA[i] < SA[top] do
4:          PSV_text[SA[top]] ← SA[top − 1]
5:          NSV_text[SA[top]] ← SA[i]
6:          top ← top − 1                                      ▷ Stack.pop()
7:      top ← top + 1                                          ▷ Stack.push()
8:      SA[top] ← SA[i]
9:  p ← 1
10: while p ≤ n do
11:     LPF ← 0
12:     PrevOcc ← 0
13:     psv ← PSV_text[p]
14:     nsv ← NSV_text[p]
15:     l ← lcp(psv, nsv)
16:     if T[p + l] = T[psv + l] then
17:         l ← l + 1
18:         LPF ← l + lcp(p + l, psv + l)
19:         PrevOcc ← psv
20:     else
21:         LPF ← l + lcp(p + l, nsv + l)
22:         PrevOcc ← nsv
23:     if LPF > 0 then
24:         OUTPUT-FACTOR((p − PrevOcc, LPF))
25:     else
26:         OUTPUT-FACTOR((T[p], 0))
27:     p ← p + max(1, LPF)
```

2.5 Variable-byte coding

Variable-byte coding is a technique for compressing non-negative integers by reducing the number of leading zeros from their binary representation [29; 24]. The technique was originally introduced as part of the MIDI file format by the name *variable-length quantity* [27]. Afterwards, the technique has been adopted under multiple different names. Some of the well-known names for this technique include *VInt*[1], *varint*[2], *varbyte* [1], and *VByte* [20; 13]. In this thesis, I use the name VByte.

VByte is a byte-aligned coding, where each byte-sized block contains seven bits of data and a *continuation* bit. The continuation bit, stored in the most significant bit of each byte, determines whether the VByte code continues in the following byte. To construct a VByte code, given a non-negative fixed-length integer, we concatenate it seven bits at a time with a continuation bit, starting from the least significant bit, until the most significant set bit. The continuation bit in the least significant byte is set, while the other continuation bits are left unset. The VByte encoded integer is then stored in big-endian byte order.[3] For example, integer 73 (1001001) encoded in VByte is $\underline{1}$1001001, and 824 (11 00111000) is $\underline{0}$0000110 $\underline{1}$0111000. Listing 2.1 contains the embodiments of VByte encoding and decoding algorithms.

By carefully studying the VByte decoding algorithm, we observe that VByte coding contains some redundancy, as it is possible to pad VByte encoded integers with bytes consisting of only zero bits. For example, VBytes codes $\underline{1}$1001001, $\underline{0}$0000000 $\underline{1}$1001001, and $\underline{0}$0000000 $\underline{0}$0000000 $\underline{1}$1001001 all correspond to integer 73. To remove this redundancy, we alter the coding so that the smallest possible $n+1$ byte code represents a value that is strictly one greater than the largest value representable with n-byte code.[4] As a result, the altered VByte coding can encode integers in $[2^{7(n-1)}+2^{7(n-2)}+\ldots+2^7, 2^{7n}+2^{7(n-1)}+\ldots+2^7)$ using $n > 1$ bytes. This is a significant improvement over traditional VByte, which can encode integers in range $[2^{7(n-1)}, 2^{7n})$ using $n > 1$ bytes. Note that the behaviour for single-byte codes remains unchanged. Both codings can encode integers in $[0, 2^7)$ using a single byte. This alteration in coding turns traditional VByte coding into a bijective

[1] Apache Lucene. https://lucene.apache.org/core/3_5_0/fileformats.html (April 13, 2022)

[2] Google Protocol Buffers. https://developers.google.com/protocol-buffers/docs/encoding (April 13, 2022)

[3] Even though VByte is usually implemented using little-endian byte order, I chose to implement it using big-endian byte order to simplify the decoding algorithm.

[4] The origin of this alteration is in *varint* coding used in *Git* version control system. https://github.com/git/git/blob/master/varint.c (April 14, 2022)

```
encode_vbyte(val):
    vbyte = (val & 0x7F) | 0x80
    val = val >> 7
    vbyte_len = 1
    while val > 0:
        vbyte = (vbyte << 8) | (val & 0x7F)
        val = val >> 7
        vbyte_len = vbyte_len + 1
    while vbyte_len > 0:
        write_byte(vbyte & 0xFF)
        vbyte = vbyte >> 8
        vbyte_len = vbyte_len - 1
    return

decode_vbyte():
    byte = read_byte()
    res = byte & 0x7F
    while byte < 0x80:
        byte = read_byte()
        res = (res << 7) | (byte & 0x7F)
    return res
```

Listing 2.1: Traditional approach to VByte encoding and decoding algorithms. `encode_vbyte(val)` takes a non-negative integer value as an argument and writes it to a byte stream as VByte code. `decode_vbyte()` reads a VByte code from a byte stream and returns an integer value.

mapping, and therefore I refer to it as *bijective VByte*. The changes in implementation of encoding and decoding algorithms are demonstrated in Listing 2.2, while the differences in the resulting codes are demonstrated in Table 2.5.

Table 2.5: Comparison of 32-bit fixed-length binary to traditional VByte and bijective VByte.

n	32-bit fixed-length binary	VByte	VByte (bijective)
0	00000000 00000000 00000000 00000000	10000000	10000000
1	00000000 00000000 00000000 00000001	10000001	10000001
2	00000000 00000000 00000000 00000010	10000010	10000010
⋮			
127	00000000 00000000 00000000 01111111	11111111	11111111
128	00000000 00000000 00000000 10000000	00000001 10000000	00000000 10000000
129	00000000 00000000 00000001 00000001	00000001 10000001	00000000 10000001
⋮			
16383	00000000 00000000 00111111 11111111	01111111 11111111	01111110 11111111
16384	00000000 00000000 01000000 00000000	00000001 00000000 10000000	01111111 10000000
16385	00000000 00000000 01000000 00000001	00000001 00000000 10000001	01111111 10000001
⋮			
16511	00000000 00000000 01000000 01111111	00000001 00000000 11111111	01111111 11111111
16512	00000000 00000000 01000000 10000000	00000001 00000001 10000000	00000000 00000000 10000000
16513	00000000 00000000 01000000 10000001	00000001 00000001 10000001	00000000 00000000 10000001

It is also possible to generalize the idea of VByte coding, including the bijective variant, to other block widths. Instead of attaching a continuation bit to every seven data bits, we attach it to every k data bits and call the coding VLQ_k, a variable-length quantity with k data bits per block. Note that VByte is a byte-aligned special case of VLQ, namely VLQ_7. General VLQ codes may be useful for improving the compression ratio when the distribution of integers is known, but this improvement usually comes with severe performance degradation due to loss of byte alignment. For high throughput, it is best to use VByte or word-aligned VLQ codes. One exception to this is VLQ_3, or *VNibble*, as it is possible to implement extraction of nibble-sized fields from a byte stream in an efficient manner. While experimenting with the prototype data compressor, I utilized exclusively the bijective variants of VByte, VNibble and general VLQ codes.

```
encode_vbyte(val):
    vbyte = (val & 0x7F) | 0x80
    val = val >> 7
    vbyte_len = 1
    while val > 0:
        val = val - 1
        vbyte = (vbyte << 8) | (val & 0x7F)
        val = val >> 7
        vbyte_len = vbyte_len + 1
    while vbyte_len > 0:
        write_byte(vbyte & 0xFF)
        vbyte = vbyte >> 8
        vbyte_len = vbyte_len - 1
    return

decode_vbyte():
    byte = read_byte()
    res = byte & 0x7F
    while byte < 0x80:
        byte = read_byte()
        res = res + 1
        res = (res << 7) | (byte & 0x7F)
    return res
```

Listing 2.2: Bijective variants of VByte encoding and decoding algorithms. The changes compared to traditional VByte algorithms are highligted.

2.6 Unary coding

Unary coding is one of the simplest *prefix coding* schemes for integers [23, Sec. 2.1]. A prefix code is uniquely decodable due to the fact that none of the codes is a prefix of another code.

Prefix coding. Let Σ be the source alphabet and Γ be the code alphabet. Code $C\colon \Sigma \mapsto \Gamma^*$ is a prefix coding if $C(a)$ is not a prefix of $C(b)$, $\forall a, b \in \Sigma$ and $a \neq b$.

There are many ways to implement Unary coding and the choice depends on one's specific needs. With minor changes, Unary coding can be used to encode either positive or non-negative integers. It is also possible to extend Unary coding to handle signedness. In this thesis, Unary coding is exclusively utilized for the variable length portion of Golomb-Rice codes introduced in the next section and therefore a simple variant which can encode non-negative integers is sufficient. The chosen variant encodes integer n as n 0-bits, followed by a single 1-bit.

$$Unary(n) = \overbrace{000\ldots0}^{\text{n zeros}}1$$

Alternatively, the bits could be reversed, but the particular encoding was chosen as it provides a possibility for a performance optimization introduced later in Section 3.7.3. Table 2.6 contains examples of Unary codes, while the encoding and decoding algorithms are embodied in Listing 2.3.

Table 2.6: Examples of Unary codes. The stop character '1' has been highlighted by separating it from the actual value with a blank character.

n	binary	Unary code
0	0	1
1	1	0 1
2	10	00 1
3	11	000 1
4	100	0000 1
5	101	00000 1
6	110	000000 1
7	111	0000000 1
8	1000	00000000 1
9	1001	000000000 1

```
encode_unary(val):
    while val > 0:
        write_bit(0)
        val = val - 1
    write_bit(1)
    return

decode_unary():
    res = 0
    bit = read_bit()
    while bit != 1:
        res = res + 1
        bit = read_bit()
    return res
```

Listing 2.3: Algorithms for encoding and decoding integers with Unary coding. `encode_unary(val)` takes a non-negative integer value as an argument and encodes it to a bit stream as Unary code. `decode_unary()` reads an Unary code from a bit stream and returns an integer value.

2.7 Golomb-Rice coding

Golomb coding is a parameterized prefix coding for non-negative integers [6; 23, Sec. 2.23]. The encoding format depends on the choice of the parameter m (modulus), which is used in encoding the integer in two parts. The first part is always variable length, but the second part depends on the choice of m. If m is a power of two, then the second part has a fixed length. Otherwise, the second part too has variable length. The special case, where m is a power of two, is called Golomb-Rice coding [21; 23, Sec. 2.24]. Even though Golomb-Rice coding might in some scenarios have an adverse effect towards encoding efficiency, its main advantage is increased simplicity, which often leads to better encoding and decoding speed.

To construct Golomb-Rice code for integer n, two quantitites q (quotient) and r (remainder) need to be computed first by

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \text{ and}$$
$$r = n - qm.$$

The encoded representation is formed by concatenating q encoded in Unary, with a fixed length binary representation of r of $\log_2 m$ bits. Table 2.7 contains examples of Golomb-Rice codes with different bases, while the encoding and decoding algorithms are embodied in Listing 2.4.

In this thesis, I use "Golomb-Rice$_k$" and "GR$_k$" to refer to Golomb-Rice coding that uses base k, corresponding to modulus $m = 2^k$.

Table 2.7: Examples of Golomb-Rice codes with different bases. The two parts of Golomb-Rice codes have been highlighted by separating the Unary encoded quotient from the fixed length remainder with a blank character.

n	binary	GR_1	GR_2	GR_3	GR_4
0	0	1 0	1 00	1 000	1 0000
1	1	1 1	1 01	1 001	1 0001
2	10	01 0	1 10	1 010	1 0010
3	11	01 1	1 11	1 011	1 0011
4	100	001 0	01 00	1 100	1 0100
5	101	001 1	01 01	1 101	1 0101
6	110	0001 0	01 10	1 110	1 0110
7	111	0001 1	01 11	1 111	1 0111
8	1000	00001 0	001 00	01 000	1 1000
9	1001	00001 1	001 01	01 001	1 1001
10	1010	000001 0	001 10	01 010	1 1010
11	1011	000001 1	001 11	01 011	1 1011
12	1100	0000001 0	0001 00	01 100	1 1100
13	1101	0000001 1	0001 01	01 101	1 1101
14	1110	00000001 0	0001 10	01 110	1 1110
15	1111	00000001 1	0001 11	01 111	1 1111
16	1 0000	000000001 0	00001 00	001 000	01 0000

```
encode_golomb_rice(val, m):
    encode_unary(val >> m)
    if m > 0:
        mask = 1 << (m - 1)
        while mask > 0:
            if (val & mask != 0)
                write_bit(1)
            else
                write_bit(0)
            mask = mask >> 1
    return

decode_golomb_rice(m):
    res = decode_unary() << m
    if m > 0:
        rem = 0
        while m > 0:
            rem = (rem << 1) | read_bit()
            m = m - 1
        res = res | rem
    return res
```

Listing 2.4: Algorithms for encoding and decoding integers with Golomb-Rice coding. `encode_golomb_rice(val, m)` takes a non-negative integer value and a non-negative Golomb-Rice base as an argument, and writes the integer value to a bit stream as Golomb-Rice code with specified base. `decode_golomb_rice(m)` takes a non-negative Golomb-Rice base as an argument, reads a Golomb-Rice code with specified base from a bit stream, and returns an integer value.

3 Methods

In this chapter, I introduce the individual building blocks of the prototype data compressors discussed in Chapter 4. First, I define some general aspects concerning the experiments discussed later in this chapter. I continue by defining a compressed file format, which facilitates compression and decompression in practice. After that, I discuss the two most computationally intensive subtasks of suffix array-based data compression, which are the construction of the suffix array and the lcp-comparisons performed during Lempel-Ziv factorization. Subsequently, I introduce four factorization algorithms with varying degrees of greediness that are based on the greedy KKP3 factorization algorithm discussed in Chapter 2. After factorization, I describe the base encoding, which provides the actual compression, and three enhancements that can be layered on top of it for additional improvements in the compression ratio. Finally, I close this chapter by describing the general principles that contribute to the performance of prototype data compressors.

3.1 General remarks

The prototype data compressor is called SALZ, which is an acronym of suffix array-based Lempel-Ziv data compressor. SALZ was developed with the C programming language, and it is compiled with GCC 9.3.0 by a CMake based build system. The compilation uses flags `-Wall`, `-Werror`, `-Wextra` and `-pedantic`[1] and optimization level `-O3`. SALZ links against the 32-bit version of the *libsais*[2] suffix array construction library, which is built using `-Wall` compilation flag and optimization level `-O3`. The experiments rely on data being processed as text consisting of byte alphabet symbols and, therefore, the input and output buffers are simple byte arrays. The extra space used by SALZ is allocated in 4-byte words, which should be considered when discussing memory usage. All asymptotic complexities refer to a variable n, the length of the text, which corresponds to the size of the uncompressed file in bytes. SALZ was developed on a system that uses little-endian byte order, and some of the advanced features and methods described in this chapter rely on that. Parallelization in the form of multithreading is not used, but some

[1] One of the experimental lcp-comparison methods in Section 3.4.1 uses additionally the `-march=native` compilation flag. However, that method was not used in any of the final compressor designs.

[2] https://github.com/IlyaGrebnov/libsais (May 6, 2022)

of the advanced techniques use instruction level parallelism or vectorised instructions for increased performance.

3.2 Compressed file format

To facilitate the decompression of compressed data at some later point, at least some metadata is needed besides the compressed payload. The prototype data compressor uses the concept of block size to compress files in segments of equal size, with the exception that the last segment may be shorter. The block size can range from 32kiB to 128MiB, with the default being 64kiB. The block size is encoded in VByte at the very beginning of the compressed data. This allows the decompression process to allocate a sufficient amount of resources for decompression, and it acts as an additional sanity check against malformatted segments.

To simplify decompression, each compressed segment is preceded by its size encoded in VByte. This allows the decompressor to process a single whole segment of compressed data at a time, therefore eliminating the need to shuffle buffers and minimizing the amount of copying of data between buffers. This means that decompression is performed exactly one compressed segment at a time. Otherwise, the decompressor would need to handle continuous decompression and have the means to request more data to complete the decompression of a segment and store data belonging to the next segment for future use.

Each compressed segment consists of streams that, like compressed segments, are preceded by their size encoded in VByte. The file format does not reach any deeper, instead the stream contents and their utilization depend solely on the chosen encoding. The streams were introduced to facilitate some of the advanced encoding methods described in Section 3.6. The layered file format is illustrated in Figure 3.1.

Besides the minimal metadata that is used by SALZ, a fully featured data compressor could benefit from additional types of metadata. In addition to the block size, the header of the file could store additional information about the compressed file. Examples of such could be a file format identifier to determine the file type, or additional information about the data compressor or encoding, to support backward compatibility if new features are added to the data compressor at some later point. Additional sanity checks in the form of segment-specific checksums would allow ensuring data integrity.

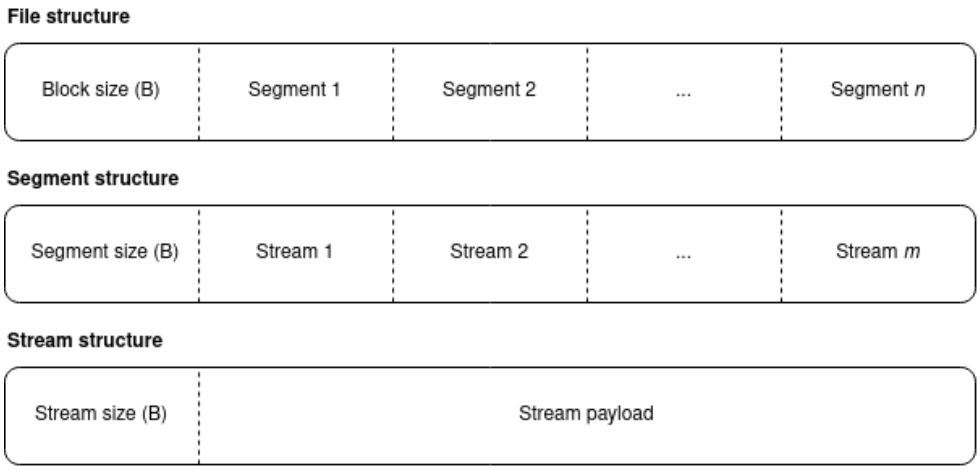


Figure 3.1: The compressed file format used by SALZ.

3.3 Suffix array construction

Suffix array construction is computationally intensive, and therefore a great deal of care should be taken when choosing a method for it. For typical stringology, the suffix array is usually constructed for the whole text. However, from the perspective of a general-purpose data compressor, constructing the suffix array of the whole file might not be feasible or even possible with large files, because of limited computational resources. Suffix array construction has been heavily studied, and there are myriad methods with different asymptotic complexities and practical differences. Since the focus of the thesis is on implementing a data compressor, I used existing methods that are readily available.

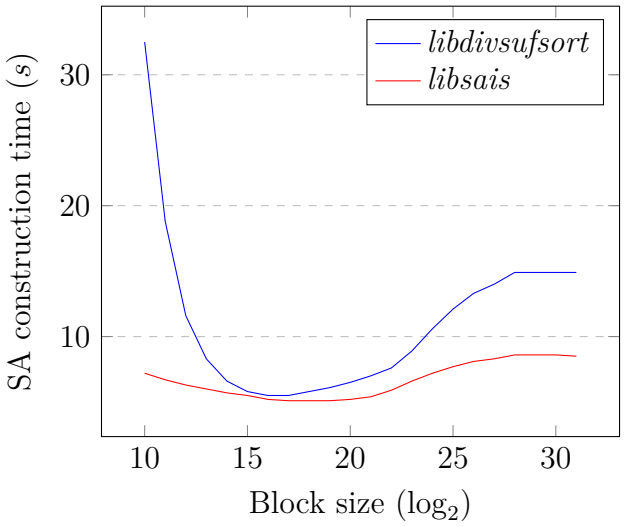
The current default choice for suffix array construction is the mature and fast *libdivsufsort*[1] library by Yuta Mori. The asymptotic running time of *libdivsufsort* is $O(n \log n)$ and it uses $5n + O(1)$ bytes of memory, which includes the memory required to store the input and output. I performed the preliminary benchmark to determine the effects of block size on suffix array construction using *libdivsufsort* with block sizes in the range 1kiB to 1GiB, in powers of two. As the benchmark payload, I used a collection of 100MB files, which are described later in Section 4.1. Based on the benchmark, I observed that the best runtimes were achieved with block sizes in the range 32kiB to 128kiB. I observed the total time spent in suffix array construction to increase as the block size diverges from the block size corresponding to the minimum time. The behaviour was further studied with the `perf` and `valgrind` programs, which revealed the phenomenon explaining the effect. When the

[1] https://github.com/y-256/libdivsufsort (May 6, 2022)

block size increases, the number of cache misses increases because of the inefficient memory access pattern of the suffix array construction algorithm. As the block size decreases, the amount of work increases with a greater number of machine instructions.

A recent competitive alternative to *libdivsufsort* in suffix array construction is *libsais*[1] library by Ilya Grebnov. The time complexity of *libsais* is $O(n)$ and it uses at most $7n$ bytes of memory.[2] It is based on the SA-IS [18] algorithm, but it uses the capabilities of modern hardware to tackle the high constants hidden by the asymptotic notation. It is considerably faster than *libdivsufsort* on modern hardware; the improvements diminishing on older hardware. The best performance with *libsais* is obtained with block sizes between 32kiB to 1MiB, the higher end yielding better results. The increase in running time as the block size increases is not nearly as dramatic as with *libdivsufsort*. The graph in Figure 3.2 illustrates the above observations on a single payload.

Figure 3.2: Suffix array construction times for varying block sizes with *libdivsufsort* and *libsais* libraries, benchmarked on *silesia* payload. Refer to Section 4.1 for more information about the payload.



Besides the suffix array construction times, I also measured the time spent in disk I/O and the mean LCP values across all segments. Disk I/O times follow a similar pattern as the suffix array construction time, with the best time observed with block sizes in the range 256kiB to 4MiB. However, I observed the disk I/O times to be insignificant when compared to the suffix array construction times. I also observed the mean LCP values to vary heavily depending on the payload. As expected, repetitive and well-compressible payloads had

[1] https://github.com/IlyaGrebnov/libsais (May 6, 2022)

[2] In addition to worst case extra memory requirement of $2n$ bytes, $5n$ bytes are required to store the input and output.

high mean LCP across all block sizes. With some individual payloads, the amount of repetition was observed to sharply rise at some specific block size, indicated by a great increase in mean LCP over a single step in block size. Some payloads showed relatively low mean LCP values across all block sizes, which seemed like a favorable scenario for trying to experiment with suffix array construction via traditional string sorting algorithms.

Because of the mean LCP observations, I also benchmarked the suffix array construction via ordinary string sorting algorithms. In the benchmark, I made use of the enormous collection of string sorting algorithms by Tommi Rantala.[1] I benchmarked all 147 sequential string sorting algorithms of the collection against *libdivsufsort*. Due to such many algorithms, I limited the block sizes to the range 64kiB to 4MiB and truncated the test payloads to 10MB. Over all payloads and block sizes, suffix array construction via string sorting was considerably slower than utilizing *libdivsufsort*. Even at its best, the construction of the suffix array via string sorting took twice the time of *libdivsufsort*. No single string sorting algorithm was superior, as the most suitable algorithm varied according to the payload and block size. With string sorting algorithms, the effect of increasing the block size was even more fatal than with *libdivsufsort*. As expected, I observed a low mean LCP to correspond to better performance with string sorting algorithms.

After experimenting with suffix array construction, I chose *libsais* as the suffix array construction method for SALZ because of its superior performance and currently active development.

3.4 Lcp-comparison

As mentioned at the start of this chapter, lcp-comparison is central to Lempel-Ziv factorization. It is used to determine the longest previous factors for factorized text positions and makes up most work performed during the post suffix array construction phase of Lempel-Ziv factorization. The amount of work grows depending on the number of factorized text positions and the number of previous positions compared per each factorized position. With greedy factorization, the number of factorized positions is a subset of all positions dependent on the repetitiveness or compressibility of the text, while with non-greedy factorization, all text positions might be factorized. When factorization is performed using a suffix array, there are at most two candidates for the longest previous factor for each text position.

[1] https://github.com/rantala/string-sorting (May 6, 2022)

A naïve (and indeed natural) approach to lcp-comparison compares two positions of a byte buffer, single byte at a time, until the first mismatching byte is found. The number of bytes compared until the first mismatching byte corresponds to the length of the longest common prefix of the two positions. Listing 3.1 embodies such a naïve lcp-comparison function.

```
size_t lcp_cmp(uint8_t *text, size_t text_len, size_t pos1, size_t pos2)
{
    size_t len = 0;
    if (pos2 < pos1)
        swap(pos1, pos2);
    while (pos2 + len < text_len && text[pos1 + len] == text[pos2 + len])
        len++;
    return len;
}
```

Listing 3.1: Naïve single byte-at-a-time lcp-comparison function.

The naïve lcp-comparison function can be sped up by performing the comparison multiple bytes at a time, in a seemingly parallel manner. On the other hand, if the positions are known to share a common prefix of some known length, it is possible to skip comparing that part and instead start the comparison from an offset, reducing the number of compared bytes. The possibility of starting the comparison from an offset depends on the availability of additional information. Since the used factorization techniques are based on the suffix array, such information is fortunately available. In this section, I describe techniques that use either platform native word size or vectorised instructions to perform lcp-comparison with words, a method to reduce the number of compared bytes that applies to the setting of non-greedy factorization, and finally observations about the performance of the two methods on their own and when combined.

3.4.1 Faster comparison with words

Most modern machines process data with a native 8-byte word size, which allows performing lcp-comparison multiple bytes at a time. A straightforward and effective approach is to augment the naïve lcp-comparison algorithm by reading and comparing 8-byte words until a first mismatching word is encountered and, at that point, revert to a single-byte comparison.

A more sophisticated approach is to make use of the `TZCNT` instruction, which counts the number of trailing zero bits in a word. After reading the two 8-byte words, instead of comparing them to each other, we compute a bitwise `XOR` between them. If the result is zero, we know the words were identical and we can continue to try the next two words. However, if the result is non-zero, we know that there is a mismatch somewhere inside the words. By computing the number of trailing zeroes in the result, we determine the number of matching bits until the first mismatching bit. Since we are interested in the number of matching bytes, we can divide the number of matching bits by 8 to determine the number of matching bytes until the first mismatching byte. We can then compute the lcp value by summing the total length of fully matched words with the number of matching bytes within the mismatching words as computed with `TZCNT`. Since we do not want to read past the allocated memory area, we need to fall back to the single-byte comparison whenever there are less than 8 bytes left until the end of the string.

Below is a demonstration of using the technique described above to compute the longest common prefix for strings $A =$ "abbbbbbb" and $B =$ "afffffff". From the final `XOR` result, we see that the trailing 10 bits are matching, which corresponds to a mismatch in the second byte, i.e., only the first byte is matching. Note that the example uses little-endian byte order.

	8 byte word			
	'a' (0x61)	'b' (0x62)		'b' (0x62)
A	01100001	01100010	...	01100010
	'a' (0x61)	'f' (0x66)		'f' (0x66)
B	01100001	01100110	...	01100110
`XOR`	00000000	00000100	...	00000100

The possibility of unaligned memory access is a concern, as in general, the factors have lengths that cause them to start or end between word boundaries. Even though most modern platforms can perform unaligned memory accesses, even without negative effects on computation cost, it might cause fatal problems including hardware crashes on some platforms.[1] And even if the platform supports unaligned memory access, it is still an undefined behaviour according to the C standard, which may cause problems depending on the compiler or chosen compiler options. One of the simplest mitigations is to make use of the `memcpy` system call, which a clever compiler can optimize in a platform-supported

[1] https://lemire.me/blog/2012/05/31/data-alignment-for-speed-myth-or-reality/ (May 6, 2022)

way. For example, with x86-64 instruction set, calling `memcpy` with a constant size of 8 bytes, will emit a single `MOV` instruction, while with ARMv5, the compiler will emit instructions that build an 8-byte word with the help of single byte moves and bit shifting. A fully portable lcp-comparison method using 8-byte words, augmented with `TZCNT` is showed fully in Listing 3.2.

```
size_t lcp_cmp8(uint8_t *text, size_t text_len, size_t pos1, size_t pos2)
{
    size_t len = 0;
    if (pos2 < pos1)
        swap(pos1, pos2);
    while (pos2 + len < text_len - 7) {
        uint64_t val1, val2;
        memcpy(&val1, &text[pos1 + len], 8);
        memcpy(&val2, &text[pos2 + len], 8);
        uint64_t diff = val1 ^ val2; // XOR
        if (diff != 0)
            return len + (__builtin_ctzll(diff) / 8); // TZCNT
        len += 8;
    }
    while (pos2 + len < text_len && text[pos1 + len] == text[pos2 + len])
        len++;
    return len;
}
```

Listing 3.2: Lcp-comparison performed using 8-byte words, augmented with `TZCNT` instruction. The method expects little-endian byte order.

On platforms supporting vectorised instructions, lcp-comparison can be performed with even larger words. The SSE2, AVX2 and AVX-512 instruction set extensions allow the use of words with sizes 16, 32, and 64 bytes, respectively. Since AVX-512 support is still quite rare, I discuss here only SSE2 and AVX2 based methods. The basic idea is quite similar to the previously described methods using 8-byte words. We first read two 16-byte words with the `MOVDQU` instruction, which supports unaligned access. After reading the words, we compare them for equality with `PCMPEQB` instruction, which produces a 16-byte comparison result. `PCMPEQB` compares the individual bytes in the corresponding positions of the words, and, in the case of equality, sets the byte in the corresponding position of the comparison result to `0xFF`, and otherwise sets it to `0x00`. We then convert the bytes of the comparison result into individual bits of a 4-byte comparison mask with the `PMOVMSKB` instruction, which extracts the most significant bit of each individual byte in

the comparison result. Finally, we use bitwise `NOT` to receive a final comparison mask, where each zero bit corresponds to a matched byte in the original word and each non-zero bit corresponds to a mismatched byte. Since only the lower half of the mask is used, the upper half needs to be unset before proceeding to the final step. The final comparison mask is similar to the final 8-byte word of the previous technique, except this time each single bit corresponds to a matched byte. If the comparison mask is zero, we continue matching with words. Otherwise, we count the trailing zeros in it to derive the number of matched bytes before the first mismatched byte. The number of matched bytes overall is then the sum of the total length of fully matched words and the last matching bytes computed with `TZCNT`. In order to not read past the allocated memory area, we again need to fall back to a single-byte comparison if there are less than 16 bytes left until the end of the string. Listing 3.3 contains an embodiment of lcp-comparison with 16-byte words performed with the help of the SSE2 instruction set extension. To convert the function to perform lcp-comparison with 32-byte words with the help of AVX2 instruction set extension, `MOVDQU`, `PCMPEQB` and `PMOVMSKB` instructions need to be replaced with their corresponding AVX2 variants `VMOVDQU`, `VPCMPEQB` and `VPMOVMSKB`. In addition, the upper half of the comparison mask must not be unset with the 32-byte version as `VPMOVMSKB` uses the whole 4-byte integer.

Below is a demonstration of using the technique described above to compute the longest common prefix for texts A = "abbbbbbbbbbbbbb" and B = "acccccccccccccc". The number of trailing zeros in the final `NOT` result indicates that only the first bytes are matching. Note that the example uses little-endian byte order, which results in a difference in ordering between `PCMPEQB` and `PMOVMSKB` results.

	16 byte word			
	'a' (0x61)	'b' (0x62)		'b' (0x62)
A	01100001	01100010	...	01100010
	'a' (0x61)	'c' (0x63)		'c' (0x63)
B	01100001	01100011	...	01100011
	0xFF	0x00		0x00
`PCMPEQB`	11111111	00000000	...	00000000
	16 bit integer			
`PMOVMSKB`	00000001	00000000		
`NOT`	11111110	11111111		

```
size_t lcp_cmp16(uint8_t *text, size_t text_len, size_t pos1, size_t pos2)
{
    size_t len = 0;
    if (pos2 < pos1)
        swap(pos1, pos2);
    while (pos2 + len < text_len - 15) {
        __m128i val1 = _mm_loadu_si128((void *)&text[pos1 + len]); // MOVDQU
        __m128i val2 = _mm_loadu_si128((void *)&text[pos2 + len]);
        __m128i cmp = _mm_cmpeq_epi8(val1, val2); // PCMPEQB
        int cmpmask = _mm_movemask_epi8(cmp); // PMOVMSKB
        int diff = ~cmpmask; // NOT
        diff &= 0x0000ffff; // Fix MSB bits
        if (diff != 0)
            return len + __builtin_ctz(diff); // TZCNT
        len += 16;
    }
    while (pos2 + len < text_len && text[pos1 + len] == text[pos2 + len])
        len++;
    return len;
}
```

Listing 3.3: Lcp-comparison performed using 16-byte words with the help of 128-bit SSE2 and `TZCNT` instructions. The method can be converted to perform comparison in 32-byte words by converting instructions to corresponding 256-bit AVX2 versions and skipping the MSB correction step. The method expects little-endian byte order.

3.4.2 Reduced number of comparisons

With greedy Lempel-Ziv factorization, the number of symbol comparisons performed by the naïve lcp-comparison algorithm is linear to the length of the factorized text. However, the minimum-cost factorization algorithms need to factorize all text positions, which increases the number of symbol comparisons to quadratic if comparisons are performed naïvely. Fortunately, the properties of the suffix array and the LPF array can be exploited to obtain an algorithm that can reduce the number of symbol comparisons to linear.

In [3], Crochemore and Ilie showed how the LPF and $PrevOcc$ arrays could be obtained in linear time using only the suffix array. Their approach relied on a key property of the LPF array, $LPF[i] \geq LPF[i-1] - 1$, to minimize the number of symbol comparisons. This property holds even when only positions corresponding to suffixes that are lexicographically smaller or greater are considered, and therefore allows the computation of full LPF and $PrevOcc$ arrays in linear time, using the PSV and NSV arrays, as described in Section 2.4.1.

Implementing the Crochemore and Ilie's approach is a simple matter of storing the lengths of the longest common prefixes associated with PSV and NSV positions for the current factorized positions and using them in the factorization of the next position. SALZ stores the needed information in a lcp-comparison context structure that is demonstrated in Listing 3.4. Additionally, the lcp-comparison function must be able to start the comparison from an offset. Listing 3.5 demonstrates a variant of the naïve lcp-comparison function, which has the capability of starting the comparison from an offset that is supplied as an additional argument. To compute the factor starting at the current position i, the algorithm takes the results of factorizing the previous position $i-1$ and decrements both of them by one. If the resulting values are positive, they are used to skip the corresponding number of bytes for lcp-comparison between positions i and both $PSV[i]$ and $NSV[i]$. Finally, the obtained results are stored in the lcp-comparison context structure, for use in the factorization of the next position $i+1$. The algorithm is demonstrated in Listing 3.6.

This technique eliminates comparing symbols that are known to match. Removing these unnecessary symbol comparisons results in asymptotically less work and practically faster comparison in all scenarios. The only exception is a text which contains no factors at all, in which case the performance remains unchanged.

```
struct lcp_cmp_ctx {
    size_t psv_len;  // lcp between previous PSV and previous
                     // factorization positions
    size_t nsv_len;  // lcp between previous NSV and previous
                     // factorization positions
};
```

Listing 3.4: Lcp-comparison context structure that stores the results of previous factorization.

```
static size_t lcp_cmp(uint8_t *text, size_t text_len, size_t common_len,
        size_t pos1, size_t pos2)
{
    size_t len = common_len;
    if (pos2 < pos1)
        swap(pos1, pos2);
    while (pos2 + len < text_len && text[pos1 + len] == text[pos2 + len])
        len++;
    return len;
}
```

Listing 3.5: Naïve single byte-at-a-time lcp-comparison function with a capability to start comparison at an offset.

```c
static void lz_factor(struct lcp_cmp_ctx *ctx, uint8_t *text,
        size_t text_len, size_t pos, int32_t psv, int32_t nsv)
{
    size_t psv_len = 0;
    size_t nsv_len = 0;

    if (psv != -1) {
        size_t common_len = max(0, ctx->psv_len - 1);
        psv_len = lcp_cmp(text, text_len, common_len, psv, pos);
    }

    if (nsv != -1) {
        size_t common_len = max(0, ctx->nsv_len - 1);
        nsv_len = lcp_cmp(text, text_len, common_len, nsv, pos);
    }

    ctx->psv_len = psv_len;
    ctx->nsv_len = nsv_len;
}
```

Listing 3.6: Lempel-Ziv factorization algorithm that minimized the number of symbol comparisons using LPF array key property $LPF[i] \geq LPF[i-1] - 1$.

3.4.3 Lcp-comparison performance

The technique to perform comparison with words can be applied to all Lempel-Ziv factorization variants, which are introduced in Section 3.5. In the case of greedy and lazy non-greedy variants, there is no need to factorize all positions. In that setting, the 8-byte word size provides the largest improvement, resulting in a 7.5%–38.6% improvement in factorization time compared to performing lcp-comparison a single byte at a time. 16- and 32-byte word sizes were measured to result in up to 36.5% and 35.4% improvements, respectively. In all cases, the amount of improvement decreases when the block size is increased. In addition, using 16- and 32-byte word sizes with the largest block size of 128MiB results in slight performance degradation.

With the minimum-cost algorithms, the improvements are even more significant as all text positions are factorized. In this setting, the best improvements were obtained by performing the lcp-comparison with a 32-byte word size, which resulted in a 78.1%–84.5% improvement in factorization time, when compared to a single-byte comparison. 16- and 8-byte word sizes resulted in 76.8%–82.2% and 73.4%–79.4% improvements, respectively.

The technique to reduce the number of compared symbols relies on the factorization of consecutive text positions, which makes it applicable mainly to minimum-cost algorithms. It could be adapted to the factorization of consecutive literal positions with greedy and lazy non-greedy variants, but I did not pursue this direction. By utilizing this technique, I observed performance improvements of 69.9%–83.9%, when compared to performing lcp-comparison a single byte at a time. The performance benefit obtained by reducing the number of compared symbols is therefore quite comparable to performing lcp-comparison with words.

When the above techniques are used in combination, we observe both improvement and degradation of performance depending on the chosen block size. With a word size of 8 bytes, I observed improvements of 13.4%–42.5% with block sizes between 32kiB and 16MiB and degradation up to 16.3% with larger block sizes up to 128MiB, when compared to a single byte-at-a-time comparison with a reduced number of symbol comparisons. A 16-byte word size resulted in a 3.6%–32% improvement and up to a 27.5% degradation with the same block sizes. Finally, a 32-byte word size resulted in a 5.0%–25.9% improvement with block sizes between 32kiB and 8MiB and degradation up to 38.7% with larger block sizes up to 128MiB.

Because of the significant increase in suffix array construction time with larger block

sizes, we would like to avoid block sizes larger than 8MiB. We can therefore neglect the degradation caused by combining these techniques with larger block sizes. By focusing only on block sizes between 32kiB and 16MiB, we obtain a performance improvement of 82.7%–87.0% over the naïve baseline implementation, by using the combination technique with platforms native word size. Therefore, I use the 8-byte word size in all SALZ variants. In addition, the minimum-cost variants use the technique to reduce the number of compared symbols for further improvement. A micro-benchmark corresponding to different lcp-comparison techniques can be found from Appendix B.

3.5 Practical Lempel-Ziv factorization

While the final compressed size depends mostly on the encoding of factor information, the qualities of factors produced by the factorization stage also play a big role. The traditional greedy approach to Lempel-Ziv factorization is optimal regarding the number of factors produced, but it rarely corresponds to optimal results regarding the compressed size. In this section, I describe four different factorization methods with varying degrees of greediness, which are all based on the greedy KKP3 factorization algorithm. I start by introducing factor pruning heuristics to the greedy KKP3 factorization algorithm and subsequently introduce a variable amount of laziness to it. In addition, I describe two factorization algorithms capable of minimum encoding cost optimization. Since the effects of these factorization algorithms depend on the encoding, I discuss the results later in Chapter 4.

3.5.1 Greedy factorization

The main problem when applying traditional LZ77 factorization to data compression is that the factorization alone does not reflect the overall encoding cost. Trying to come up with an efficient encoding for plain LZ77 factorization is difficult as with any encoding, fixed or variable length, the encoded lengths of short factors can easily exceed the length of the original data. Because of this, LZSS factorization is generally a more suitable approach as it enhances the traditional LZ77 factorization with the distinction of literals from factors. However, even though LZSS significantly reduces this adverse behaviour, it does not entirely eliminate it.

One of the most often used strategies to mitigate the above drawbacks is to limit the

maximum factor offset, which can further be used to limit the minimum factor length. By adding an upper bound to the factor offset, the length of its fixed length representation can be bounded too. Ideally, the upper bound should be a power of two for encoding efficiency. The upper bound of the encoded factor offset can then be used to determine the lower bound for the factor length. The bound should be set to the first factor length, that, with the given factor offset restriction, can provide an encoded factor which uses less space when compared to encoding consecutive literals. Pruning the factors in this way results in an increased number of factors and generally mandates the use of a sliding window factorization algorithm to maintain a better compression ratio.

With suffix array-based Lempel-Ziv factorization, using a sliding window is not a practical option as dynamic updates to the suffix array come with high cost. Emulating a sliding window by constructing the suffix array of a block that is larger than the usable window is generally a weak option too. Because the suffix array-based Lempel-Ziv factorization algorithm provides factors that are the lexicographically closest, the probability that a factor is found outsize the usable window increases with the block size. It is therefore necessary that the encoding must be able to represent factor offsets up to the suffix array block size. The best scenario is therefore to use a variable length encoding for factor offsets. This, in turn however causes inconvenience when determining the lower bound for the factor lengths. If the minimum factor length is set too low, short factors with large offsets end up having encoded representations that use more space compared to consecutive literals. On the other hand, if the lower bound for the factor length is too high, the data compressor might end up encoding consecutive literals instead of a short factor with a small offset, which would have a more compact encoded representation. Therefore, with variable length representations, factors should also be pruned based on either a heuristic or by comparing the actual encoded lengths.

The approach that is used in SALZ is to always choose the longer of the factor candidates, or, in the case of factors with equal length, the candidate that has a smaller offset. The encoded length of the candidate factor's offset is then compared to the actual factor length. Since, with LZSS-based encoding, a factor length is a sufficiently close approximation to the cost of consecutive literals, factors with offsets greater than or equal to the factor length can be discarded. The greedy variant of SALZ is strongly based on the KKP3 factorization algorithm and it diverges from it only by having a lower bound of 3 for factor length and by additionally pruning the factors based on their offset. Based on a benchmark, both increasing and decreasing the lower bound set for factor length lowered

the compression ratio.

Discarding a factor causes the data compressor to emit a literal and immediately continue factorization at the next position. Since the upper bound for the factor offset is defined by the block size, there is a constant upper bound for the discardable factors length. Therefore, the pruning does not cause an increase in asymptotic time complexity. Asymptotic complexity remains $O(n)$ and space usage is $12n + O(1)$ bytes, as with the original KKP3 factorization algorithm. Even though the factorization variant is called greedy, it is in fact non-greedy due to not using all produced factors. It is, however, as greedy as is practical.

3.5.2 Lazy non-greedy factorization

With greedy factorization, we only use a factor if it saves space compared to encoding the occurrence as consecutive literals. Moving further into the non-greedy factorization, we concern ourselves with the possibility that using a future factor occurring to the right of the current position would provide us with a better compression ratio than the factor at the current factorization position. This consideration is not novel and has been used both in practice and discussed in scientific literature [8; 22, Sec. 6.25]. Before introducing the technique used in SALZ, I first describe the two antecedent methods from which I derived the technique.

The first of the methods appears in the well-known and widely used `gzip` program. During factorization, before emitting a factor, `gzip` checks if there is a longer factor available at position immediately to the right of the current position [22, Sec. 6.25]. If a longer factor is found, `gzip` emits a literal, followed by the longer factor, instead of emitting the original factor. The second one is a generalization of the `gzip` method with a predefined amount of laziness, introduced by Horspool in [8]. Horspool's method relies on the notion that skipping multiple positions (rather than just a single one) might be profitable. It uses a constant-sized "look-ahead buffer" in which it looks for a factor that extends past the current factor. If it finds such, and combining it with a partial application of the current factor would result in a more succinct encoded representation, the encoder emits the original factor as shortened, followed by the longer factor instead of the original factor in its full length. In practice, the Horspool method, with a look-ahead buffer of size 1, corresponds to the `gzip` method.

The weakness of the `gzip` method is that it checks only the next position for each factor and therefore it cannot use factors occurring further to the right. The major strength

is that checking only one adjacent position adds only a little extra computation to the factorization. On the other hand, Horspool's method checks a constant number of positions for every factor, which, depending on the number of checked positions, might not be a significant compression improvement over the `gzip` method or could increase the amount of computation significantly. SALZ strives to benefit from checking multiple positions without increasing the amount of computation significantly by utilizing the `gzip` approach in a lazy manner. In case we find a longer factor at the next position, we emit a literal and instead of committing to the longer factor, we continue the factorization from the next position. The negative impact is that the compressor ends up sometimes emitting consecutive literals, which might result in larger encoding than a shortened factor would have. However, the positive impact is that SALZ can improve factors in a continuous manner beyond the next position with a minimal increase in the amount of computation. The asymptotic time complexity can be kept in $O(n)$ by using the technique to reduce the number of symbol comparisons during factorization, which was introduced in Section 3.4.2. The increase in used extra space is $O(1)$ as only a constant number of new local variables are introduced.

3.5.3 Minimum-cost factorization

We can achieve even better results with an algorithm that computes the total cost to encode a text with a predefined encoding. I found the initial inspiration towards minimum-cost optimization while comparing SALZ to LZ4[1] and its high compression variant LZ4HC. LZ4HC can produce an encoding with a significantly greater compression ratio than LZ4 at the cost of significantly increased computation time. Study of the source code reveals that when the level of compression increases, the number of compared positions increases and finally, with LZ4HC, a cost optimization heuristics are included.

The key notion in the minimum-cost heuristic of SALZ is that for each position of the text, there are at most three available transitions to right. For each position, with the exception of the last one, there is a transition with a length of 1 byte, corresponding to a literal. Additionally, there are at most two factor transitions with length greater than equal to 1 byte due to using the suffix array-based Lempel-Ziv factorization algorithm. As a result, a directed acyclic graph (DAG), with unique source and sink vertices, can represent the text and all available transitions. The symbols of the text correspond to vertices of the DAG and all literals and factors form the edges of it. Additionally, the

[1] https://lz4.github.io/lz4/ (May 6, 2022)

DAG is already in topological order, as each transition leads to the right in the text. In principle, the approach is similar to one introduced by Ferragina et al. in [5]. However, the advantage of SALZ's approach is that costly pruning of transitions is not needed, as the suffix array-based approach to factorization itself limits the number of transitions.

With fixed encoding, we can assign costs to all transitions by defining a cost function. Now the matter of computing the minimum cost to encode the text can be determined by simply computing the shortest path from the source to sink with the edge weights given by the cost function. Such an algorithm iterates over all vertices of the DAG in topological order and assigns each adjacent vertex a minimum cost to arrive from the current vertex. Additionally, by storing the transitions corresponding to the minimum costs, the factorization corresponding to the minimum encoding cost can then be backtracked.

In practice, the produced encoding is not necessarily globally optimal, but locally optimal, given the constraints of the suffix array-based factorization algorithm. The globally optimal factorization does not necessarily use the full length of the factors, instead the optimal solution could be to use only parts of the factors. Additionally, the algorithm only considers at most two longest previous factors, when the optimal factorization might result from using a slightly shorter factor with a smaller offset. However, pursuing a globally optimal encoding would result in a dramatic increase in running time and in the worst case, the algorithm would have an asymptotic time complexity of $\Omega(n^3)$ as it would need to check all possible factors for all lengths less than or equal to their full length for all text positions.

Deriving a cost function for an LZSS-type encoding is relatively simple. For each literal or factor, there is a corresponding flag that uses a single bit. The literals are simple byte constants, of 1 byte each. For factors, we need to compute the actual space usage of encoding the offset and length, which with integer codings used by SALZ is possible in constant time. If a block format is used, e.g., LZ4 encoding, the cost associated with the constant-sized tokens should be associated with the factors as the factors could present back-to-back and each factor potentially breaks up a run of consecutive literals.

The minimum-cost factorization separates the factorization stage from the encoding stage and further divides it into two substages. After computing the *PSV/NSV* array, we in an interleaved manner factorize all positions while updating the minimum costs from left to right. Next, we backtrack (right to left) the factorization corresponding to the minimum encoding cost. Finally, we emit the encoding corresponding to the factorization produced in the previous step from left to right. For each position, we evaluate at most

three different transitions using a constant time cost function. Additionally, we use the technique to minimize the number of symbol comparisons in the factorization, introduced in Section 3.4.2, which allows us to perform all stages with $O(n)$ time complexity. Besides the *PSV/NSV* values needed in the factorization stage, we need to store three additional values for each vertex: the cost to encode the transition, the offset used in the transition, and the current vertex. Later, in the backtracking stage, we no longer need the *PSV/NSV* values and overwrite them with the factor offsets and lengths that correspond to the minimum encoding cost. Therefore, we can implement the algorithm using $20n + O(1)$ bytes of extra space.

3.5.4 Dynamic programming minimum-cost factorization

A more refined observation about the problem is that it has an *optimal substructure* property, which allows the use of a dynamic programming approach. The optimal substructure property means that the optimal solution can be obtained using the optimal solutions of its subproblems. Regarding the factorization problem, it essentially means that if the factorization corresponding to the minimum-cost encoding from some vertex u to some other vertex v includes a transition starting from vertex s, then the optimal solution is a combination of optimal solutions from u to s and and from s to v.

The algorithm is essentially a reversed version of the previously introduced minimum-cost factorization algorithm. Instead of computing the cost to arrive from the source vertex, we compute the cost to reach the sink vertex instead. We achieve this by iterating the vertices from right to left and for each vertex, computing the cost of reaching the sink node with a literal and available factor candidates, and choosing the transition that minimizes the encoding cost to reach the sink vertex.

One benefit of a dynamic programming approach is that separate stages for computing the minimum costs and determining the optimal solution are no longer needed. Instead, a single right-to-left pass suffices for computing both the costs and determining the optimal factorization. Another benefit is that the number of memory writes is reduced by a multiplicative factor of three. In both approaches, the number of memory reads is essentially the same as for each vertex at most three costs are compared. However, with the normal approach, up to three values associated with up to three different positions are updated for each processed vertex. With the dynamic programming approach, the cost is only updated for the current vertex, which also improves the locality of memory access.

It is possible to implement the dynamic programming approach in $12n + O(1)$ bytes of extra space. We can achieve this by overwriting the suffix array with minimum-cost values and repurposing the *PSV/NSV* array for the factor offsets and lengths corresponding to the minimum encoding. Note that the repurposing of *PSV/NSV* array is only possible because of the optimal substructure property and factorization direction of the dynamic programming approach. The caveat of this approach is that we can no longer use the technique to reduce the number of symbol comparisons as the LCP array property exploited in it is not applicable in the reverse direction. In the worst case, this increases the asymptotic time complexity to $O(n^2)$.

To achieve $O(n)$ complexity, the factorization needs to be performed first, in a separate stage. We first factorize all positions from left to right, and store the offsets and lengths for both factor candidates. Then we compute the minimum costs from right to left, as explained previously. As we need to store at most two factor candidates for each text position, the extra space consumption goes back to $20n + O(1)$ bytes. However, because of more efficient cache usage and improved memory access pattern with a smaller number of writes, the dynamic programming approach routinely outperforms the normal approach.

3.6 Encoding

From the perspective of data compression, Lempel-Ziv factorization is only a preprocessing stage. The actual saving in the space consumption is a consequence of judicious encoding. On the other hand, no single encoding works in all scenarios and it must fit together with the chosen factorization method. In this section, I describe the base encoding used in SALZ and additionally three techniques that can augment it.

3.6.1 Encoding format

The starting point for experiments in encoding was a simple block format encoding that used VByte. The compressed segment consists of blocks, a single block consisting of a literal run followed by a factor. The length of a literal run is encoded in VByte at the beginning of a block. In case the length of a literal run is non-zero, it is followed by the literals. Factor offset and length are then encoded in VByte at the end of the block. The last block of the compressed segment may be incomplete, in which case it only contains a literal run. I found the compression ratios achieved with this baseline solution to be

extremely bad, as a great number of bytes are wasted by encoding the values in VByte due to generally short literal runs and factors.

To discover a more suitable encoding, I decided to determine some of the practical differences between suffix array-based Lempel-Ziv factorization and the more traditional approach of chained hashing. The most obvious way to determine this was to copy the encoding used by a third-party data compressor. The group of well-known and widely used data compressors relying solely on Lempel-Ziv factorization is small, and from that group LZ4 is the most widely adopted candidate. LZ4 is mostly known for its extreme compression and decompression speed, but it also has a high compression variant, LZ4HC, that uses the same format. LZ4HC provides satisfactory results regarding compression ratio.

LZ4 compresses files in frames[1], which consists of blocks[2] using a 64kiB sliding window. Similar to the baseline VByte encoding, LZ4 block consists of a literal run followed by a factor. The block starts with a single byte token that is divided into two nibble-sized fields. The most significant nibble represents the length of a literal run, while the least significant nibble represents the factor length. If the length of a literal run cannot be represented with a nibble-sized field, the remaining length is encoded after the token and is followed by the literals. The factor offset is encoded as a 2-byte unsigned integer in little-endian byte order after the literals. If the factor length could not be represented with a nibble-sized field, the remaining length is encoded after the factor offset. The additional length fields are encoded as a sum of consecutive byte values, which is demonstrated in Listing 3.7. Because of performance optimizations, the LZ4 block format forces the last block of a frame to contain only a literal run. In my experiments, I allowed the last block to be incomplete, but did not force it.

```
write_add_len(len):
    while len >= 0xFF:
        write(0xFF)
        len = len - 0xFF
    write(len)
```

Listing 3.7: Linear encoding used to encode the additional literal run length and factor length fields of LZ4 block format.

SALZ with a greedy factorization seems to generally outperform LZ4, but is bested by LZ4HC on some payloads. LZ4 uses a greedy hash table-based approach in factorization,

[1] https://github.com/lz4/lz4/blob/dev/doc/lz4_Frame_format.md (April 27, 2022)
[2] https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md (April 27, 2022)

while LZ4HC uses *Morphing Match Chains*[1] together with non-greedy cost optimization heuristics. Direct comparison between LZ4 and the prototype data compressor is not very useful because of significant differences in the factorization. However, based on the comparison, I could deduce that with suffix array-based factorization, the encoder should always be able to encode factor offsets up to the block size. When the block size is increased, it becomes increasingly probable that the factorization algorithm produces factor candidates with offsets too great to be used. Removing the factor offset limitation and encoding the factor offsets in VByte led to significant improvements in compression ratio for larger block sizes, which confirms the previous notion. This notion also led to the observation that emulating a sliding window factorization algorithm deteriorates the compression ratio when utilized together with suffix array-based factorization. Therefore, it is clear that with suffix array-based factorization it is not possible to compress a file as a continuous stream or a series of dependent segments, but instead the compression must be performed a single segment at a time. This characteristic of suffix array-based factorization makes the LZ4 style of encoding useless for SALZ.

Before further experiments, I collected factor statistics to help in experimental design. One profitable observation from analyzing the statistics was that with suffix array-based factorization, the runs of literals are very short on average. Therefore, moving from a literal run-based block format to a flag-based format with single byte literals proved beneficial for the compression ratio in general. The format used by SALZ is a straightforward adaptation of the LZSS type of encoding. An unset flag corresponds to a single byte literal while a set flag corresponds to a factor.

After experimenting with multiple ways of encoding the factors with the help of VByte, VNibble, Elias-Gamma, and Golomb-Rice codes, I found an encoding providing the overall best compression ratio in the block size range 32kiB to 128MiB. The encoding appears to be a local optimum, as any minor modification leads to a slight to moderate deterioration in the compression ratio. The encoding emits the factor offsets in two parts, fixed and variable length. The fixed part corresponds to the least signifying byte of a factor offset. The remainder of the factor offset is then encoded as VNibble, which in practice means that the encoded representation of a factor offset uses 12–36 bits. We encode the factor length using Golomb-Rice code with base 3. We encode both factor offset and length as biased values, with the amount of bias corresponding to the minimum values. The minimum offset is 1, since an offset of 0 would correspond to no offset. I chose the minimum factor

[1] https://fastcompression.blogspot.com/p/mmc-morphing-match-chain.html (April 27, 2022)

length to be 3, which forces shorter factors to be encoded as literals.

We write the bit flags and variable length integer codes in an interleaved manner to 8-byte fields. Literals, fixed parts of the factor offsets, and the bit fields as interleaved form the primary compressed data stream. We read and write literals and fixed parts of factor offsets synchronously, while the bit fields are buffered. At the beginning of compression, we allocate space for the first bit field at the beginning of the primary stream. When the buffered bit field is full, we write it to the previously allocated location, empty the buffer and reserve the space for the new bit field at the current output position. After compressing a segment, we fill the unused part of the buffer with zeros before flushing it. The decompression works in reverse. In the start, we fill the bit field buffer from the beginning of the primary stream and each time it gets empty, we refill it from the current input position.

For further performance improvements, the format could be enhanced by storing the flags separated from the Golomb-Rice codes. This would open up a possibility to determine the number of consecutive literals using the `LZCNT`[1] intrinsic and perform copying of literals multiple bytes at a time. Another useful observation is that since every factor has a non-negative number of literals preceding it, the space used by LZSS flags is equivalent to using the concept of literal runs and encoding their length using Unary encoding. Then, with non-repetitive payloads, swapping Unary encoding with another variable length integer code, e.g., Elias-Gamma, could improve the compression ratio even further. However, I did not pursue these directions in the prototyping of SALZ.

3.6.2 Factor offset reuse

Analyzing the factor statistics further revealed that the factor offsets have a moderately flat distribution along the whole range, which explains the failures encountered when applying variable length encoding to them. Collecting additional statistics about recently used offsets revealed that identical offsets have a minor tendency to appear shortly after their previous use. I collected the statistics with the help of a constant-sized self-organizing cache that keeps the offsets ordered from the most recently used to the least recently used. The statistics showed that there is some amount of offset reuse on all payloads. The offset reuse strongly depends on the payload and the amount of it varies dramatically. The statistics on factor offset reuse for a single payload can be found from Appendix B.

[1] Leading Zero Count.

Even though offset reuse opens up the possibility of skipping offset encoding altogether, implementing it is problematic. In LZSS encoding, single bit flags are used to distinguish between single byte literals and ordinary factors. To augment LZSS encoding with offset reuse, we need some additional information to distinguish the two different factor types. Optionally, we could add extra information to offset reusing factors to distinguish between the previous offset, second-to-previous offset, and so on up until some constant boundary. In practice, adding even a single bit of information to every factor to detect its type inflates the size of the compressed segment more than is possible to save by reusing the offsets. Augmenting the primary stream with offset reuse proved therefore not beneficial.

However, I found a profitable way to implement the offset reuse using a secondary stream. The secondary stream stores the ordinals of the factors that use the previous offset. Every time we encounter a factor with repeated offset, we write the ordinal of that factor to the secondary stream and skip the writing of the factor offset to the primary stream. The decoding process works in reverse. At the beginning of decoding, we read the ordinal of the first factor with offset reuse from the secondary stream. When it is time to decode that particular factor, we then use the previous offset instead of reading one from the primary stream, read the ordinal of the next factor with offset reuse from the secondary stream, and continue the decoding process. For decoding to work, we must store a special stop value at the end of the secondary stream to signal that there are no factors with offset reuse left. SALZ implements this by storing a value that is strictly greater than the last factor ordinal.

For efficient representation, we must encode the ordinals in the secondary stream with a variable length encoding instead of a fixed length representation. Since SALZ at this point already used multiple integer coding schemes, I experimented with what was already available. Since the values stored in the secondary stream are positive and in strictly increasing order, I layered delta encoding on top of the variable length encoding. Delta encoding replaces the actual values with differences between adjacent values. Since we access the values in a strictly sequential order, delta encoding adds only a constant amount of extra computation to decode the values. After some brief experimentation, I found VNibble encoded deltas to provide satisfactory results.

Since the problem of encoding positive and strictly increasing integers is a well-researched topic, e.g., in search engines, I could validate the optimality of the chosen encoding. One of the encodings devised to tackle such problems is Elias-Fano encoding [28]. Elias-Fano encoding is *quasi-optimal*, which means that it is close to information-theoretical lower

bound. In addition, there exists a well-defined closed form upper bound for the number of bits needed to represent a series of positive and strictly increasing integers with it. When I compared the actual bit utilization of the chosen encoding to the upper bound defined for Elias-Fano encoding, I observed that the chosen encoding performed well. In most cases, the chosen encoding could outperform the upper bound of Elias-Fano, with the encoded representation using up to 41.1% less bits in the best case. For some individual payloads with larger block sizes, the upper bound of Elias-Fano encoding in bit utilization was exceeded, with at most 6.6%.

Since the technique touches only the final encoding stage, it can be used with all factorization variants of SALZ. The extra memory needed is $O(1)$ since only a couple of new variables need to be introduced. However, there is a penalty on the compression performance. I observed the technique to provide 0.42%–0.78% improvement in compression ratio with 6.7%–16.1% increase in encoding time and 1.9%–7.7% increase in decoding time. A micro-benchmark corresponding to this technique can be found from Appendix B.

Augmenting the minimum-cost optimization heuristic with the technique is more complex. The simplest way is to ignore the encoding cost of the factor offset that is reused. To achieve that, the minimum-cost stage needs to store the offset of the factor with the minimum cost to arrive at each position. Since the minimum-cost factorization algorithm already does that to allow backtracking the minimum-cost factorization, the extra memory cost is only $O(1)$. However, since the factor offset dependencies are directed to the right, it is not possible to take factor offset reuse into account with the dynamic programming approach to the minimum-cost heuristic. Augmenting the minimum-cost heuristic with factor offset reuse adds a performance penalty as it increases the nested branching inside the minimum-cost calculation loop. Most of the penalty could be mitigated with the technique to reduce the branching introduced later in Section 3.7.4. The calculation of real costs associated with offset reuse seemed so complex that I did not pursue it any further.

The compression ratio could be improved further by using some constant number of recently used offsets in addition to the previous offset. With greedy factorization, the extra memory cost would still be in $O(1)$, but with minimum-cost factorization it would be $kn \log n$ bits, where k is the number of recently used offsets. Because of the complex nature of the technique and high memory cost, I left this direction unpursued too.

3.6.3 Incompressible segments

Some payloads can contain incompressible segments, in which case the compressed representation results in using more space than the original uncompressed data. Incompressible segments are encountered usually with small block sizes, especially when the data appears sufficiently random or has a low degree of repetitiveness. SALZ uses a simple technique to minimize the size inflation caused by incompressible segments. After compressing a segment, we compare its size to the size of the original uncompressed segment. If the compressed segment ends up using more space than the original uncompressed data, we empty the compressed streams and write the original uncompressed data behind the last stream. The added overhead of the empty streams is 9 bytes per stream, so the overall overhead caused by the incompressible segment is negligible. The greatest benefit of the approach is that it effectively minimizes the negative effect of incompressible segments without affecting the functional properties of the data compressor. We could further reduce the stream overhead of incompressible segments by adding more complexity to the encoding and decoding processes. However, since the possible gains are extremely low, I did not pursue this direction further.

3.6.4 Choosing optimal integer codes

Statistics gathered on factors showed that the structure and repetitiveness of the payload influences the distribution of factor lengths to a great degree. Factor lengths follow a distribution which rises sharply to a peak at very low factor lengths and then settles down rapidly, forming a long tail distribution. The height and position of the peak varies between payloads, with repetitive payloads having a blunter peak and a thicker tail, and non-repetitive payloads having a more prominent peak with a thinner tail. Furthermore, the position of the peak varies between payloads, so that the peak of repetitive payloads corresponds to a higher factor length compared to non-repetitive payloads. To some minor extent, factor offsets show similar behaviour even though they generally have a relatively flat distribution across a variety of payloads. The offsets generally are over the whole possible range, but for small block sizes, there is a tendency to slightly favour the smaller side. However, as the block size increases, the distribution flattens. This variance across payloads presents a possibility of tuning the parameters of the utilized variable length integer codes in the hope of increased compression ratio. Since SALZ processes files in independent segments, it is possible to tune the parameters on a per segment basis.

The base encoding utilizes Golomb-Rice codes with base 3, as I found them to be the best overall fit across a variety of payloads. However, the optimal base varies highly between payloads and even more so between individual segments of a payload. A straightforward way to choose a suitable base is to collect the occurrences of each factor length in a histogram during factor selection. Afterwards, the optimal base is the one that minimizes the cost to encode all factor lengths. Due to the previously observed distribution of the factor lengths, it is easy to see that there exists an optimal base $k = a$ such that the cost to encode all factor lengths increases when k diverges from a in either direction. Therefore, the optimal base can be determined by computing the total cost to encode all factor lengths for $k = 0, 1, 2, \ldots$, until the cost stops decreasing, at which point the optimal base corresponds to $k - 1$. Since we must know the base for decompression, we precede each compressed segment with the chosen base encoded in VNibble. Since the maximum factor length of a segment can be arbitrarily large, we should only iterate the histogram up to some predefined upper bound. In practice, I observed an upper bound in the range 1024–8192 to provide satisfactory results as a lower value affected the compression ratio negatively, and the gains in compression ratio diminished rapidly when the value was increased. There should also be a predefined upper limit on the base as otherwise we encounter an infinite loop if a segment consists only of literals, or all factors are longer than the upper bound chosen for the histogram iteration. SALZ uses 8192 as the upper bound for histogram iteration and 27 for the maximum base, which means that the overhead increase for a single segment is in $O(1)$ and the total increase in overhead depends on the number of segments. The optimization provides up to 2.1% improvement in compression ratio with an overhead increase of $10\mu s$–$100\mu s$ per segment. A micro-benchmark corresponding to this technique can be found from Appendix B.

Since the method alters the encoding, we cannot use it with factorization variants that rely on fixed encoding or make assumptions about it. This means that the method cannot be used with either of the minimum-cost factorization variants due to circular dependency, in which the chosen encoding depends on factor selection, which depends on the minimum-cost heuristic, while the minimum-cost stage depends on encoding. Integrating the method with greedy and lazy non-greedy factorization variants is straightforward, but comes with a caveat. Since the encoding can be only chosen after the factor selection is done, it is no longer possible to perform factor selection and encoding in an interleaved manner. Instead, we must divide them into stages separated by the integer code optimization stage, which increases the processing time even though the asymptotic amount of work performed remains unchanged. The extra memory usage is in $O(1)$, since for both greedy and lazy

non-greedy factorization variants, we can repurpose the space reserved for the suffix array for the factor length histogram.

For the factor offsets, I found the method to be not useful. The idea was otherwise similar, but instead of tuning the base of Golomb-Rice codes, the plan was to replace VNibble with a general VLQ. The plan had two major caveats. First, due to the relatively flat distribution of factor offsets, the iteration depth of the factor offset histogram cannot be bounded, but the whole histogram has to be iterated. Second, I found VNibble, corresponding to VLQ_3, to be optimal or near optimal for most of the segments across a variety of payloads. The gains in compression ratio were almost non-existent and the increase in processing time was too high to be justified.

3.7 Performance optimizations

The performance of a data compressor depends mostly on the algorithms chosen for the different stages of compression, but the minor implementation details of those algorithms matter as well. In this section, I introduce a variety of well-known optimization techniques that were utilized in implementing the algorithms. The optimization techniques rely on code and memory area organization, and on the use of special compiler intrinsics. Microbenchmarks corresponding to the techniques introduced in this section can be found from Appendix B.

3.7.1 Memory reuse

The number of dynamic memory area allocations and frees is rarely considered when designing a program. Since the underlying platform processes the dynamic memory allocations via system calls, it is not possible to predict their performance. On some platforms, the overhead of allocating and freeing memory may depend on the size of the memory area, and even the overhead of freeing and allocating memory areas of the same size could vary. On some platforms, freeing and allocating a memory area of the same size results in reusing that same area with reduced overhead,[1] while on some other platforms, the overhead might be unaffected.

[1] The platform used in the development of SALZ had memory management system that reused the same memory areas when memory resources were freed and allocated between individual segments. Therefore it was not possible to measure the benefits of this strategy in a meaningful manner.

In Section 3.6.1, I described how SALZ compresses arbitrarily sized files using identically sized blocks. To process each block, the compressor needs sufficiently sized input and output buffers, along with sufficient auxiliary space to store the interim values used by the chosen algorithm variant. For a large file compressed with a small block size, this could mean a significant number of calls to memory allocation and free. As previously mentioned, the size of all blocks is identical with the exception of the last block, which may be smaller than the preceding blocks. Therefore, the needed memory resources can be bound from above to process any block, which makes it possible to reuse the same memory areas for all blocks during compression of a single file.

Instead of allocating memory resources individually for each block, SALZ employs memory preallocation to eliminate the overhead of repeated memory allocation and freeing. After the block size is determined, before proceeding to actual compression, SALZ allocates all needed memory resources based on the chosen block size and combines them into an encoding context structure. During compression, SALZ keeps reusing those preallocated resources and frees them only after compressing all segments. The above applies also to decompression, with its own corresponding memory requirements. By reusing preallocated memory, the number of memory allocations to compress or decompress a file is always constant, which effectively removes the possible platform-dependent overhead caused by repeated allocation and freeing of memory. On the test platform used in this thesis, I measured allocation and freeing of memory resources for a single segment to take $0.1\mu s$–$133\mu s$ depending on the size of the segment.

3.7.2 Memory access patterns

Suffix array-based Lempel-Ziv factorization is inherently memory bound and does not include significant amounts of computation. Modern CPUs use hardware prefetching for soon to be used data if they can detect the underlying memory access pattern. In practice, CPUs can detect linear memory access in either ascending or descending order of memory addresses, which allows prefetching of data to CPU cache. Memory bound programs often get significant improvements when the memory access patterns are optimized in a way that allows hardware prefetching. Another beneficial optimization is to group the data in a way that allows fetching data that is used together using a single fetch. For example, optimizing the dynamic programming variant of the minimum-cost algorithm in a way that is described later in this section improves compression time by up to 14.7%.

In the KKP3 factorization algorithm, the first step after computing the suffix array is the computation of PSV and NSV values. In this phase, the memory access pattern for the suffix array is inherently linear. The suffix array values are read linearly and even though the already processed part of the suffix array is used as a stack, the memory access pattern is always either strictly ascending or descending. However, the access pattern for the PSV and NSV values is essentially random, which in the worst case could lead to $2n$ cache misses. Goto and Bannai were the first ones to recognize this issue and proposed the interleaving of the PSV and NSV values into a single array as a solution [7]. Since we always access PSV and NSV values at the same positions during the computation of the values, the interleaving of the arrays could effectively eliminate half of the cache misses. In the factorization stage, the benefit of interleaving is not as dramatic as we access the PSV and NSV values strictly linearly. However, as a result, the CPU prefetching has to deal with only one array, which could have a positive impact. In the greedy and non-greedy variants of SALZ which are directly based on the KKP3 factorization algorithm, the *PSV/NSV* array is stored in an interleaved manner as inspired by Goto and Bannai. Regarding the memory access patterns of greedy and non-greedy variants of SALZ, there seems to be no more space for further optimization. Fortunately, the minimum-cost variants of SALZ offer more room for the optimization of memory access patterns.

Similar to greedy and non-greedy variants, the minimum-cost algorithm starts by computing the suffix array and then proceeds to compute the interleaved *PSV/NSV* array. During the left-to-right factorization and minimum-cost computation stage, we store three interim values for each text position as explained previously in Section 3.5.3. During the right-to-left backtracking stage, those interim values are used in computing the optimal factorization, which is also stored as interim values. In the last stage, we emit the optimal encoding with a single left-to-right pass over the stored factors. During all previous stages, we access the values by the same array indices, and therefore can benefit from interleaving of them. Additionally, the algorithm can make use of two sufficiently sized arrays for reuse in all stages, instead of having separate auxiliary arrays for each algorithm stage. Before compression, we allocate two arrays: a larger and a smaller one of $12n$ and $8n$ bytes, respectively. We store the suffix array at the beginning of the larger array, followed by storing the interleaved *PSV/NSV* values in the smaller array. We then repurpose the larger array for the values computed in the forward stage of the minimum-cost factorization, which are stored in an interleaved manner. In the backtracking stage, we repurpose the smaller array for the interleaved factor offsets and lengths. This array

utilization strategy effectively minimizes the needed auxiliary space and provides benefits from the interleaving of the values accessed together.

With the dynamic programming variant of the minimum-cost algorithm, we again allocate two arrays, but this time with sizes $4n$ bytes and $16n$ bytes. We store the suffix array in the smaller array and the interleaved *PSV/NSV* values in the larger array, so that there are two array slots of padding between each *PSV/NSV* value pair. After this point, we no longer use the smaller array, but keep reusing the larger one. In the forward factorization stage of the dynamic minimum-cost algorithm, we compute the factors associated with the *PSV/NSV* pairs one by one and store them in an interleaved manner. At any given time during this stage, the already processed part of the array contains the factor offset and length pairs associated with that position, while the unprocessed part contains the untouched *PSV/NSV* pairs. Since there are two factors with their corresponding offset and length, there no longer is any padding in the already processed part of the array. In the right-to-left minimum-cost stage, the best factor candidate is determined and stored with the associated encoding cost to the end of the text in an interleaved manner. Since this time there are only three values, we need to separate the triples with a single array slot of padding. During this stage, the already processed part of the array contains the optimal factors associated with the encoding cost up to the end of the text, while the unprocessed part of the array contains the factors produced by the previous step. Finally, in the last step, we iterate over the optimal factors while emitting the encoding. Similar to before, the auxiliary space utilization is minimized and all values accessed together are interleaved. Above array reutilization strategy has the caveat of using padding, but eliminating it would lead to using more auxiliary space and shuffling multiple arrays, or allocating and freeing memory between the different stages of algorithm. However, since we want to avoid memory allocations between segments, we also want to avoid them during the processing of a single segment.

3.7.3 Integer code optimizations

The most advanced version of SALZ, the minimum-cost algorithm with factor offset reuse, utilizes a wide variety of integer codes, including VByte, VNibble and Golomb-Rice. In addition, Golomb-Rice utilizes Unary coding internally. Since the utilized codes are bit-oriented, except for VByte, it makes sense to write specialized functions to handle their specific needs. A bare minimum is to read and write multiple bits at a time with the help of bit shifting and masking, but sometimes further optimizations are possible. With VNibble,

we always read data from the input stream exactly 4 bits at a time. With Golomb-Rice$_3$, we read a variable length Unary code first, followed by exactly 3 bits. Although we could have one general function for reading any amount of bits from the input stream, it is also possible to hasten the functions operating on integer codes by having specialized functions that always read exactly the needed amount of bits. This allows reducing the amount of computation, as it is possible to define bit shift widths and bit masks as constant values, instead of (re)computing them on each function call. This optimization is applicable also for writing bits to the underlying byte stream.

For some special cases, there are even faster methods than reading or writing a constant number of bits from the input stream. Unary coding is an example of such as, as each code consists of a variable number of zero bits followed by exactly one non-zero bit. The traditional implementation reads zero bits, one bit at a time, until a first non-zero bit is encountered. The approach is problematic, especially with large Unary encoded integers, as the encoded size grows fast proportional to the magnitude of the integer. Therefore, it is useful to read and write zero bits using full bytes or words if possible. Since the underlying encoding format stores the bit stream using 8-byte words, we should also perform the Unary encoding in that capacity. For writing the integers, we write the zeros in as large chunks as possible, at most 8 bytes at a time, until we have written the whole integer. This is possible with the help of bit shifting, as is demonstrated in Listing 3.8. For reading the integers, we read zero-valued 8-byte words until we encounter the first non-zero word. We then extract the last zero bits with the help of `LZCNT`[1] instruction, as is demonstrated in Listing 3.9.

SALZ performs all bit-oriented I/O multiple bits at a time and uses specialized functions instead of generic functions whenever applicable. I observed the optimizations described here to provide 13.7%–28.7% improvement in the encoding stage and up to 13.4% improvement in the decoding stage.

3.7.4 Reduced branching

Modern processors implement instruction level parallelism [25, Ch. 16 and 18]. Instruction level parallelism must not be confused with concurrency, as true concurrency involves multiple threads executing individual workloads on a set of CPU cores or individual CPUs. Instead, instruction level parallelism involves a single thread on a single core or processor

[1] Leading Zero Count.

```c
struct io_stream {
    uint8_t *buf;      // Byte stream
    size_t buf_len;    // Length of byte stream
    size_t buf_pos;    // Current byte stream position
    uint64_t bits;     // Bit buffer
    size_t bits_avail; // Number of unused bits in bit buffer
    size_t bits_pos;   // Position allocated for bit buffer
};

static void write_zeros(struct io_stream *stream, size_t count)
{
    while (count != 0) {
        if (stream->bits_avail == 0) {
            memcpy(&stream->buf[stream->bits_pos], &stream->bits, 8);
            stream->bits = 0;
            stream->bits_avail = 64;
            stream->bits_pos = stream->buf_pos;
            stream->buf_pos += 8;
        }

        size_t write_count = min(stream->bits_avail, count);
        stream->bits <<= write_count;
        stream->bits_avail -= write_count;
        count -= write_count;
    }
}
```

Listing 3.8: Function that writes zero bits in at most 8-byte chunks to a bit stream that is interleaved with a byte-oriented output stream.

```c
static size_t read_zeros(struct io_stream *stream)
{
    if (stream->bits_avail == 0) {
        memcpy(&stream->buf[stream->buf_pos], &stream->bits, 8);
        stream->buf_pos += 8;
        stream->bits_avail = 64;
    }

    size_t ret = 0;

    while (stream->bits == 0) {
        ret += stream->bits_avail;
        memcpy(&stream->buf[stream->buf_pos], &stream->bits, 8);
        stream->buf_pos += 8;
        stream->bits_avail = 64;
    }

    size_t last_zeros = __builtin_clzll(stream->bits); // LZCNT
    stream->bits <<= last_zeros;
    stream->bits_avail -= last_zeros;

    return ret + last_zeros;
}
```

Listing 3.9: Function that reads zero bits in at most 8-byte chunks from a bit stream that is interleaved with a byte-oriented input stream.

executing multiple instructions in a seemingly parallel manner. It is achieved via instruction pipelining, which essentially means dividing instructions into smaller atomic subunits, which can be executed at the same time using different parts of the instruction pipeline.

In ideal conditions, when the instruction stream consists of operations without dependencies and no branching occurs, the instruction pipeline can be kept full and the CPU can complete a single instruction per clock cycle.[1] Dependencies between instructions and branching can significantly degrade the operation of the pipeline. The dependencies cause wait cycles in parts of the pipeline, therefore postponing the execution of future instructions. Branching is even more costly, as it leads to jump instructions, which cause the whole pipeline to be halted, emptied and rebuilt from scratch. The severity of the degradation depends on the length of the pipeline. Since only a single instruction can be fed to the pipeline per clock cycle, the pipeline will be throttled for the equivalent number of clock cycles. Modern processors utilize pipelines with dozens of stages.[2]

Since branching affects instruction throughput in such an expensive way, processor manufacturers employ a multitude of techniques to predict the outcome of branching and execute the instructions of a predicted branch in an eager manner. If a branch prediction is correct, the operation of the pipeline is not affected. In the opposite case, the pipeline is flushed and the results of eager computation are discarded. Even though branch prediction strategies in modern CPUs are quite advanced, branches taken with equivalent probability are still impossible to predict reliably. The only viable solution is to write the code without branching when it is possible and beneficial.

Usually, branches exist for a reason and we cannot simply remove them without altering the functionality of a program. However, sometimes we can rewrite functions in a branchless way with the help of bitwise operations and predication. Bitwise operations are mostly used for converting branching to computation, while predication corresponds to conditional instructions, whose effect depends on the state of the CPU flag registers.[3]

In different variants of SALZ, there are multiple places where it is possible to convert branches to computation by utilizing bitwise operations. One such place is the backtrack-

[1] In reality, modern processors have multiple execution ports that allow a throughput of multiple instructions per clock cycle. Without loss of generality, we can assume the CPU to have a single execution port.

[2] In reality, most modern processors utilize even smaller atomic instruction units, micro-operations, with even longer pipelines. Without loss of generality, we can expect the CPU to not utilize micro-operations.

[3] Some simple examples of predication include CMOV (Conditional Move), SETE/SETZ (Set if equal / if zero) and SETNE/SETNZ (Set if not equal / if not zero) instruction.

ing stage of the minimum-cost variant of SALZ, which is embodied in Listing 3.10. Before the backtracking stage, the algorithm has computed the optimal factor for each position of the input text. In the backtracking stage, it then processes those optimal factors in reverse order to determine the optimal factorization for the text. For each factorized position, the algorithm starts by computing the factor corresponding to the transition to that position. It then stores the factor for future use and, to simplify the later encoding stage, distinguishes literals from factors by setting the factor offset and length as zero for them. The function is written in the most straightforward way, with the help of branching.

```
1  void backtrack(size_t src_len) {
2      for (size_t src_pos = src_len; src_pos > 0; ) {
3          int32_t prev_pos = get_pos(src_pos);
4          int32_t prev_offs = get_offs(src_pos);
5          int32_t factor_len = src_pos - prev_pos;
6
7          if (factor_len != 1) {
8              store_offs(prev_pos, prev_offs);
9              store_len(prev_pos, factor_len);
10         } else {
11             store_offs(prev_pos, 0);
12             store_len(prev_pos, 0);
13         }
14
15         src_pos = prev_pos;
16     }
17 }
```

Listing 3.10: Example of the backtracking step of the minimum-cost variant of SALZ. For each factor in the optimal factorization, the algorithm checks if the factor length is non-zero. If the factor length is non-zero, it stores the factor length and offset for future use. Otherwise, it zeroes the factor length and offset to signal the encoding stage to emit a literal instead of a factor.

In the corresponding machine code in Listing 3.11, we see two different labels, .L2 and .L3, corresponding to branches. The beginning part of .L2 (lines 1–10) corresponds to the instructions executed at the start of the loop, before branching. Then the condition inside the `if` clause is computed (line 11) and depending on the result, the execution either jumps to .L3 (line 12) or continues from the next instruction (line 13). If no jump is made, the CPU will execute the first branch (lines 13–17), which stores the actual offset and length values. The loop condition is then checked (line 18) and, based on the result, the program jumps back to the beginning of loop .L2 (line 19) or exits from the function

(lines 20–23). If a jump to the `else` branch is made, the CPU will continue to execute the second branch (lines 25–30) which stores zeros as offset and length values. Then again, the loop condition is tested (line 31) and based on the result, a jump to the beginning of loop .L2 (line 32) is made or the function exits (lines 33–36). Since looping itself requires jumping, it is only possible to eliminate the jumps resulting from branching.

Listing 3.12 demonstrates a branchless version of the example function, which has exactly the same behaviour as the branching one. The loop again begins with extracting and computing the information needed to determine the correct factor offset and length. However, this time the algorithm does not have two branches, but it makes use of the result of the original `if` clause condition in arithmetic. In C, the condition `factor_len != 1` evaluates to integer 0 or 1, depending on the factor length being equal to 1 or not. The algorithm then uses the result of evaluating the condition as a multiplier for the factor offset and length, which, with a literal, leads to storing them as zeroed and otherwise storing them as unmodified.

The corresponding machine code in Listing 3.13 now has only a single label, namely .L14. The beginning of .L14 (lines 1–10) corresponds to fetching the needed values and computing the factor length. Then the algorithm evaluates the condition (line 11), computes and stores both the factor offset (lines 12–16) and the factor length (lines 17–20). Finally, the algorithm tests the loop condition (line 21) and either jumps to the beginning of loop .L14 (line 22) or exits (lines 23–27).

When we compare the branchless version of the function with the original one, we observe that the algorithm performs some extra arithmetic operations instead of jumping between branches. Since predicting a loop condition is trivial,[1] the pipeline now only needs to be flushed when the loop is complete. The branchless version of the function should now be able to keep the pipeline operational, with the instruction throughput being dependent only on the dependencies between the instructions. In this example, converting branching to branchless computation improves the time spent in the function by 4.7%–22.7%. It is, however, good to remember that branchless functions are not beneficial in all cases. When branchless code is used in places where the CPU branch predictor could predict branching correctly in a repeatable and reliable manner, the extra computation associated with branchless code will most likely cause a negative effect on performance. The key to achieving performance gains with this method relies, therefore, on robust benchmarking

[1] In our application, the loop condition is always `true` for all iterations except the one that finally breaks the loop.

```
.L2:
        mov     rdi, rbx
        mov     r12d, ebx
        call    get_pos(unsigned long)
        mov     rdi, rbx
        mov     ebp, eax
        call    get_offs(unsigned long)
        sub     r12d, ebp
        movsx   rbx, ebp
        mov     esi, eax
        cmp     r12d, 1
        je      .L3
        mov     rdi, rbx
        call    store_offs(unsigned long, int)
        mov     esi, r12d
        mov     rdi, rbx
        call    store_len(unsigned long, int)
        test    rbx, rbx
        jne     .L2
        pop     rbx
        pop     rbp
        pop     r12
        ret
.L3:
        xor     esi, esi
        mov     rdi, rbx
        call    store_offs(unsigned long, int)
        xor     esi, esi
        mov     rdi, rbx
        call    store_len(unsigned long, int)
        test    rbx, rbx
        jne     .L2
        pop     rbx
        pop     rbp
        pop     r12
        ret
```

Listing 3.11: Assembly code corresponding to contents of the `for` loop of Listing 3.10. For each factor, the algorithm checks if the factor length is non-zero and jumps to the branch corresponding to appropriate action is performed.

```
void backtrack_branchles(size_t src_len) {
    for (size_t src_pos = src_len; src_pos > 0; ) {
        int32_t prev_pos = get_pos(src_pos);
        int32_t prev_offs = get_offs(src_pos);
        int32_t factor_len = src_pos - prev_pos;

        store_offs(prev_pos, (factor_len != 1) * prev_offs);
        store_len(prev_pos, (factor_len != 1) * factor_len);

        src_pos = prev_pos;
    }
}
```

Listing 3.12: Branchless variant of Listing 3.10 that results in exactly the same result. However, instead of comparison driven branching, the function uses the result of a comparison as a multiplier to compute the values to be stored for later use in encoding stage.

and profiling.

3.7.5 Compile time branch prediction

In the previous section, I showed how branchless code can improve the performance when a set of values is computed differently depending on certain condition. However, in some scenarios, the functionality of each branch differs so greatly that it is not possible to write branch-free code. This is especially true when the branches execute other functions or cause side effects. Fortunately, many modern C compilers still provide a way for mitigating the increased overhead caused by branching.

Listing 3.14 embodies a simple example of a function `foo` that takes an integer argument `x`. If the argument is zero, the function will return the result of function `bar`. Otherwise, if the argument is non-zero, the function will return the result of function `baz`. Listing 3.15 embodies the corresponding machine code after compilation, which shows that for each non-zero argument, an extra jump instruction is executed, provided that the CPU did not succesfully predict the correct branch. The misprediction of the correct branch causes the CPU pipeline to be flushed and subsequently rebuilt, which causes instruction throughput to decrease.

Now suppose that from some specific knowledge (perhaps gathered by profiling the program), we know that the probability of `x` being non-zero is significantly higher than

```
.L14:
        mov     rdi, rbx
        mov     r13d, ebx
        xor     ebp, ebp
        call    get_pos(unsigned long)
        mov     rdi, rbx
        mov     r12d, eax
        call    get_offs(unsigned long)
        sub     r13d, r12d
        movsx   rbx, r12d
        cmp     r13d, 1
        mov     rdi, rbx
        setne   bpl
        imul    eax, ebp
        mov     esi, eax
        call    store_offs(unsigned long, int)
        mov     esi, ebp
        mov     rdi, rbx
        imul    esi, r13d
        call    store_len(unsigned long, int)
        test    rbx, rbx
        jne     .L14
        pop     rbx
        pop     rbp
        pop     r12
        pop     r13
        ret
```

Listing 3.13: Assembly code corresponding to contents of the `for` loop of Listing 3.12. As expected, the algorithm executes the same exact machine instructions for all of the factors.

```
int foo(int x)
{
    if (x == 0)
        return bar();
    else
        return baz();
}
```

Listing 3.14: Example function `foo()` that takes an integer argument `x`. In case the argument is zero, the function will return the result of function `bar()`. Otherwise, it will return the result of function `baz()`.

```
foo(int):
        test    edi, edi
        jne     .L2
        jmp     bar()
.L2:
        jmp     baz()
```

Listing 3.15: Machine code corresponding to Listing 3.14. For every non-zero argument, extra jump instruction is performed.

the probability of it being zero. For this kind of scenario, GCC provides an intrinsic `__builtin_expect`,[1] which allows the programmer to provide branch prediction in the source code that is later used in the organization of the code flow. The intrinsic `__builtin_expect(long exp, long c)` takes two arguments, `exp` for an expression to be evaluated and `c` for the expected outcome of evaluating the expression. In Listing 3.16, the original condition `if (x == 0)` has been substituted with the expectation that `x` is non-zero, i.e., with `if (__builtin_expect(x == 0, 0))`. From the corresponding machine code after compilation in Listing 3.17 we can observe that the program flow has been reversed. With a non-zero argument, the program no longer has to execute the extra jump instruction.

```
int foo(int x)
{
    if (__builtin_expect(x == 0, 0))
        return bar();
    else
        return baz();
}
```

Listing 3.16: Function `foo()` provided with the expectation that argument `x` is non-zero.

In SALZ, there are exactly two logical places for this kind of optimization. The first one is the optimization that allows factor offset reuse through a separate stream of factor ordinals introduced in Section 3.6.2. Based on statistics gathered during implementing that feature, the factor offset reuse is possible only for a fraction of factors. Therefore, it is possible to provide the compiler with an expectation that favors normal factors instead of the ones that reuse the previous offset for both encoding and decoding. In principle,

[1] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (May 10, 2022)

```
foo(int):
        test    edi, edi
        je      .L4
        jmp     baz()
.L4:
        jmp     bar()
```

Listing 3.17: Machine code corresponding to Listing 3.16. Due to reversed program flow, extra jump instruction is no longer needed for a non-zero argument.

providing this expectation information could mitigate some of the overhead caused by factor offset reuse. However, in practice, the change further increases the overhead, which highlights the need to quantify the results through benchmarking.

The second possibility for utilizing compile time branch prediction is the optimization that allows the compressor to choose the optimal integer codes for the factor lengths introduced in Section 3.6.4. That technique chooses an optimal base for the Golomb-Rice codes used to encode the factor lengths. The special case of Golomb-Rice$_0$ corresponds to Unary coding and because of its rapidly increasing codeword lengths, it is used only infrequently. It is therefore possible to optimize the functions related to Golomb-Rice by providing an expectation that Golomb-Rice codes are to be encoded with a non-zero base and opt for utilizing specialized Unary coding functions for Golomb-Rice codes with a zero base. Here, providing expectations results in reducing the increased overhead by up to 8.7%.

3.7.6 Loop unrolling

Sometimes, it is possible to reduce the adverse effects of branching by manually unrolling loops. A standard implementation of VByte, VNibble and general VLQ codes relies on looping. Listing 3.18 contains an example of function `vbyte_size`, that returns the number of bytes needed to represent an integer using VByte. However, when the integers can be limited to some known maximum value, we can unroll the loop manually for faster performance.[1] With SALZ, the integers are limited by the utilized word size, which limits the unsigned integers from above to a maximum value of $2^{32} - 1$. The performance gain results from the CPU branch predictor getting accustomed to the distribution of integers and, therefore, being often able to correctly predict the outcome of branching,

[1]The optimization is inspired by the great *libvbyte* library by Christoph Rupp, which utilizes loop unrolling with VByte. https://github.com/cruppstahl/libvbyte (May 10, 2022)

which minimizes the number of CPU instruction pipeline flushes. Listing 3.19 contains an unrolled version of the `vbyte_size` function that relies on the distribution of the encoded integers being skewed towards smaller values. If the distribution was skewed toward larger values, the clauses could be reversed and, if the distribution was mostly uniform, the function could be changed to perform an unrolled binary search for the correct branch. With SALZ, the values are on the smaller side, so I implemented the optimization as shown. It is possible to implement the unrolling with encoding and decoding too, but with decoding, the performance gains seem non-existent because in between comparisons, more data needs to be read from the byte stream.

```
size_t vbyte_size(uint32_t val)
{
    size_t len = 1;
    while ((val >>= 7) > 0) {
        val -= 1
        len += 1
    }
    return len;
}
```

Listing 3.18: A standard loop-based implementation of a function that returns the number of bytes needed to represent an integer using VByte.

```
size_t vbyte_size(uint32_t val)
{
    if (val < (1 << 7))
        return 1;
    if (val < (1 << 14) + (1 << 7))
        return 2;
    if (val < (1 << 21) + (1 << 14) + (1 << 7))
        return 3;
    if (val < (1 << 28) + (1 << 21) + (1 << 14) + (1 << 7))
        return 4;
    return 5;
}
```

Listing 3.19: Unrolled variant of the function that returns the number of bytes needed to represent an integer using VByte. Note that the compiler will substitute all computed values in this example with constant values during the compilation.

In SALZ, manual unrolling of loops was implemented for all functions related to VByte

and VNibble codes. According to a benchmark, loop unrolling can result in up to 6.5% improvement in the combined performance of the stages using those codes. In SALZ, this includes the minimum-cost optimization and encoding stages.

3.7.7 Copying in words

Decompressing a file compressed with SALZ is very simple and leaves very little space for optimization after the integer code optimizations described in Section 3.7.3. The input stream is processed in a tight loop, which, with a literal copies a single byte from the input stream to the output stream and with a factor, copies a previous occurrence from some earlier position of the output stream to the end of it. However, the copying of factors contains a major bottleneck with significant impact on decompression performance.

For copying chunks of memory, `memcpy` and `memmove` system calls are usually the best options. They have high performance that results from optimizations that exploit features of the underlying platform, e.g., memory alignment correction combined with word-sized copying, possibly even with the help of vectorised instructions. With self-referential factors, i.e., when the factor length is greater than the factor offset, such system calls cannot be used as the copied memory area is only partially initialized at the time of invoking the call. With self-referential factors, we must perform the copying in a loop that copies at most the factor offset number of bytes in a single iteration, which in the worst case is a single byte at a time.

Even though the above technique might look profitable, it rarely improves performance when compared to the most straightforward way of copying all factors a single byte at a time. This is because the compiler cannot further optimize variable-sized calls to `memcpy` or `memmove`, and with a sufficiently large offset, copying a single byte at a time allows lots of room for instruction level parallelism to provide increased instruction throughput. The logical next step is then to perform the copying in some constant-sized chunks instead of a single byte at a time, the platform native word size of 8 bytes being the most obvious choice. Copying a word at a time is significantly faster than copying a single byte at a time, and as a bonus it still leaves room for benefitting from instruction level parallelism. In case the factor offset is greater than or equal to 8 bytes, there are no problems and we can simply perform the copying using words. This means that we might write up to 7 bytes of extra data to the output stream, which we must consider when writing to the end of the buffer. However, factors with offsets less than 8 bytes pose a challenge. Fortunately,

the factor offset can always be increased to 8 bytes by performing additional steps before the copying occurs.

A generic algorithm to increase small factor offsets to 8 bytes involves three steps.[1] In the first step, we copy the first 4 bytes of a factor a single byte at a time, to facilitate offsets of less than 4 bytes. Then we copy the next 4 bytes from a position that is dependent on the factor offset. Finally, we adjust the copy position according to the factor offset. If the factor length is less than or equal to 8 bytes, there is no need to perform any additional copying. Otherwise, we copy the rest of the factor using 8-byte words. Since the copy positions in the second step and the copy position adjustments in the final step are constants specific to each factor offset, we can store them in a look-up table to avoid computing them. Listing 3.20 contains a function that illustrates the above technique.

Alternatively, we could perform the copying using 4-byte word size or even larger words of 16, 32 or 64 bytes with the help of vectorised instructions. A word size of 4 bytes has the benefit of simplifying the offset correction, but it leads to a smaller increase in performance when compared to an 8-byte word size. The larger word sizes have an undesired effect of increasing the complexity of offset correction instead. Additionally, the factors are short on average and the vectorised instructions have high latency. Therefore, the word size of 8 bytes seems to be the optimal choice for a general factor copying method, resulting in 13.3%–25.2% improvement in decoding time.

[1]The origins of this algorithm are in LZ4 source code, while the origins of the technique, in general, are unknown. https://github.com/lz4/lz4 (April 27, 2022)

```
1  static const int snd_copy_pos[8] = { -1, 0, 0, 1, 0, 4, 4, 4 };
2  static const int src_ptr_inc[8] =  { -1, 0, 0, 2, 0, 3, 2, 1 };
3
4  static void cpy_factor(uint8_t *buf, size_t pos, size_t offs, size_t len)
5  {
6      uint8_t *src = &buf[pos - offs];
7      uint8_t *dst = &buf[pos];
8      uint8_t *end = dst + len;
9
10     if (offs < 8) {
11         dst[0] = src[0];
12         dst[1] = src[1];
13         dst[2] = src[2];
14         dst[3] = src[3];
15         dst += 4;
16         memcpy(dst, src + snd_copy_pos[offs], 4);
17         src += src_ptr_inc[offs];
18         dst += 4;
19     }
20
21     while (dst < end) {
22         memcpy(dst, src, 8);
23         dst += 8;
24         src += 8;
25     }
26 }
```

Listing 3.20: Example function that copies factors in 8-byte words, with factor offset correction for factors with offsets smaller than the minimum copy size. The function is adopted from LZ4 source code with minor (simplifying) modifications.

4 Results

In this chapter, I describe how SALZ was benchmarked, specify the benchmarked data compressor designs built with the techniques from the previous chapter, and summarize the benchmark results.

4.1 Benchmark payload collection

As mentioned in the previous chapter, I gathered a set of statistics on the characteristics of KKP3 factorization algorithm on different payloads before starting the development of SALZ. Initial ideas about possible directions of development were devised while studying those statistics, but even at that point, it was clear that the development setting was to be highly experimental, carried out mostly by trial and error. Using the trial-and-error method relies on being able to produce reliable observations on the studied phenomenon. Therefore, one of the first steps preceding the actual development was building a carefully curated benchmark payload collection.

A variety of corpora for compression benchmarking already exist. *The Calgary Corpus*[1] and *The Canterbury Corpus*[2] are two of the most well-known corpora, but they are old and therefore do not represent the current data compression payloads. In particular, the individual payloads in those corpora are too small (< 1 MiB) to provide any useful observations on modern hardware. *The Silesia compression corpus*[3] is a more recent compression payload collection that better represents the needs of modern data compression. It exhibits a good degree of variety, but the individual payloads in it are still fairly small (< 49 MiB). An even more recent take is *The Pizza&Chili Corpus*[4], which is a text collection compiled especially for research on both compressed and uncompressed text indexes. The individual payloads in it represent the workloads used widely in academia and they are of sufficient size. Therefore, I chose *The Pizza&Chili Corpus* as the base of my benchmark payload collection.

[1] https://web.archive.org/web/20061209055036/http://links.uwaterloo.ca/calgary.corpus.html (May 10, 2022)

[2] https://corpus.canterbury.ac.nz/descriptions/ (May 10, 2022)

[3] http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia (May 10, 2022)

[4] http://pizzachili.dcc.uchile.cl/ (May 10, 2022)

The Pizza&Chili Corpus contains a variety of files between 53.24MiB and 2.05GiB. Since I wanted to benchmark often and have the benchmarks complete in moderate time, I truncated each file to 100MB. However, I felt that *The Pizza&Chili Corpus* alone was not diverse enough, so I augmented the collection with payloads that better represent those typical in the industry. Besides *The Pizza&Chili Corpus*, I added XML formatted text in the form of a data dump from English Wikipedia, source code in the form of a recent Linux kernel, a full concatenation of *Silesia* corpus and truncated parts of it, and an empty file consisting only of zero bytes. The produced benchmark payload collection is diverse enough to judge the behaviour of the data compressor for both academic purposes and the needs of the industry.

The individual files included in the benchmark payload collection, descriptions of their contents, and URLs for obtaining them are listed below.[1]

- The *Pizza&Chili* corpus:[2]
  - *dblp.xml8*: First $100,000,000$ bytes of the XML dataset, which consists of XML formatted "bibliographic information on major computer science journals and proceedings". (σ: 97; H_0: 5.228 bits) http://pizzachili.dcc.uchile.cl/texts/xml/
  - *dna8*: First $100,000,000$ bytes of the DNA dataset, which contains "a sequence of newline-separated gene DNA sequences". (σ: 16; H_0: 1.977 bits) http://pizzachili.dcc.uchile.cl/texts/dna/
  - *english8*: First $100,000,000$ bytes of the ENGLISH dataset, which contains a "concatenation of English text files [without headers]". (σ: 239; H_0: 4.556 bits) http://pizzachili.dcc.uchile.cl/texts/nlang/
  - *pitches*: The PITCHES dataset ($55,832,855$ bytes), which contains "a sequence of pitch values". (σ: 133; H_0: 5.628 bits) http://pizzachili.dcc.uchile.cl/texts/music/
  - *proteins8*: First $100,000,000$ bytes of the PROTEINS dataset, which contains "a sequence of newline-separated protein sequences". (σ: 27; H_0: 4.190 bits) http://pizzachili.dcc.uchile.cl/texts/protein/

[1] A precompiled collection can also be downloaded from https://www.cs.helsinki.fi/u/akiutosl/bench_collection.tar.gz.

[2] Alphabet sizes (σ) and empirical entropies of order 0 (H_0) are quoted from http://pizzachili.dcc.uchile.cl/texts.html on April 22, 2022.

  - *sources8*: First 100,000,000 bytes of the SOURCES dataset, which consists of "C/Java source code ... [of] linux-2.6.11.6 and gcc-4.0.0 distributions". (σ: 230; H_0: 5.540 bits) http://pizzachili.dcc.uchile.cl/texts/code/

- Additional payloads:

  - *enwik8*: First 100,000,000 bytes of the English Wikipedia dump from Mar. 3, 2006. The dataset consists of "UTF-8 encoded XML consisting primarily of English text". http://mattmahoney.net/dc/textdata
  - *kernel8*: First 100,000,000 bytes of *Linux 5.11.11* source code in an uncompressed tar archive. https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.11.11.tar.gz
  - *silesia*: The contents of *Silesia compression corpus* concatenated together in alphabetical order. Silesia compression corpus consists of various files of different data types "between 6MB and 51MB". http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia
  - *silesia8[h/m/t]*: First/middle/last 100,000,000 bytes of contents of *silesia* payload.
  - *zero8*: 100,000,000 zero bytes.

4.2 Benchmark environment and procedure

All benchmarking was performed on a *Lenovo ThinkPad X1 Carbon Gen 6* equipped with *Intel(R) Core(TM) i7-8550U* CPU running at 1.80GHz, 16GB of DDR3 memory running at 2133MHz and *Western Digital WD Black SN750 M.2 NVMe* SSD. The underlying operating system was Ubuntu 20.04.3 LTS with Linux 5.11.0-37, GCC 9.3.0 and GLIBC 2.31.

SALZ by default outputs the compression ratio and total compression and decompression times, including disk I/O. If built with `-DENABLE_STATS` switch, it also outputs the total time spent in different stages of compression. The additional measurements include the times spent on suffix array construction, *PSV/NSV* array computation, Lempel-Ziv factorization, minimum-cost optimization and encoding. I used these more refined time measurements mainly to produce micro-benchmarks needed to evaluate the performance of the individual techniques and optimizations described in Chapter 3. The procedure itself is a simple Bash script comprising the steps listed below.

1. Checkout the SALZ variant selected for benchmarking from the Git repository.
2. Initialize the build environment.
3. Build the SALZ variant selected for benchmarking.
4. Benchmark the selected SALZ variant with one or more benchmark payloads, with all block sizes between 32kiB and 128MiB.
   (a) Use SALZ to compress the payload.
   (b) Use SALZ to decompress the compressed file produced in the previous step.
   (c) Verify that the decompressed file produced in the previous step matches with the original payload.

For the benchmarks to represent real-life scenarios and to produce reliable results, I took the following measures in benchmarking.

- Different storage medias have different I/O speeds and thus all reading and writing must be performed on the same storage media to produce comparable results. In addition, the benchmark must be performed on an actual block device, as the use of a RAM disk or similar could provide overly optimistic results. Therefore, I performed all SALZ benchmarks using the SSD mentioned earlier in this section.
- In real-life scenarios, a single payload is rarely compressed many times in a row with different block sizes, nor is a compressed file decompressed immediately after compressing it. That approach jeopardizes the reliability of the benchmark as fragments of the files could be available for use in file system buffers or block device cache. Fortunately, Linux file system buffers can be flushed with `sync` command and block device caches (page cache, dentries and inodes) with `echo 3 > /proc/sys/vm/drop_caches` command. In SALZ benchmarking, I ran these two commands before every compression and decompression command to make sure that all data was written to the block device and read from it.
- To produce benchmark time measurements that are comparable, we must use a reliable way of measuring time. SALZ uses the `clock_gettime` system call from `time.h` which is a POSIX implementation of high-resolution timers. I used `clock_gettime` with `CLOCK_MONOTONIC` clock type, which represents the monotonic time since the system was last booted and is not affected by discontinuous jumps in the system time.

- The *Intel(R) Core(TM) i7-8550U* CPU utilized in benchmarking has 4 physical cores that are capable of hyper-threading. This means that each physical core can execute two threads at the same time using shared resources. From the perspective of the operating system, there are 8 logical CPUs to be used in computation. The Linux scheduler exhibits natural CPU affinity by executing processes on the same CPU "as long as it is practical for performance reasons".[1] In practice, the benchmark process might be forced to migrate to another CPU when resuming from sleep caused by another process being executed on the same CPU. With the `taskset` utility, the benchmark process can be bound to a specific CPU, which prevents the process from being migrated. However, binding the benchmark process to a specific CPU is not enough as it does not prevent other processes executing on the same CPU. In addition, another process being executed on the sibling CPU might affect the performance due to shared physical resources. Fortunately, the Linux kernel has the `isolcpus` boot option, which prevents processes being executed on the specified CPUs unless explicitly requested with `taskset` command. In SALZ benchmarking, I used `isolcpus=0,4` boot option to prevent processes from being executed on the first physical CPU core and `taskset -c 0,4` command to bind the benchmark process to that otherwise disabled core.[2] This way, I could reserve a single physical CPU core exclusively for the benchmark process.

- I observed the results right at the beginning of the benchmark session to provide overly optimistic performance, which was indicated by a monotonic decrease in performance over the first few iterations. To mitigate this effect, the actual benchmarks were preceded by a sufficient number of "warmup" iterations for which the results were discarded. With SALZ, I observed 5 warmup iterations with 1MiB block size to eliminate the undesired behaviour.

- Even when using all the above methods, the benchmark results still fluctuated slightly between iterations. Therefore, I executed each benchmark setting for 5 iterations and produced the final benchmark result by taking the mean of the results.

[1] `TASKSET(1)` - https://man7.org/linux/man-pages/man1/taskset.1.html (May 11, 2022)

[2] The mapping between physical cores and logical CPUs depends on the hardware and platform. With Linux, the mapping can be verified from `/proc/cpuinfo`.

4.3 Final data compressor designs

For the final comparison, I built 7 different data compressor designs, each a different combination of the techniques introduced in Chapter 3. All designs utilize the file format described in Section 3.2 and *libsais* library for suffix array construction. The lcp-comparison is always performed with the 8 byte word size as described in Section 3.4.1. In addition, all minimum-cost designs utilize the technique to reduce the number of symbol comparisons described in Section 3.4.2. The utilized encoding is the base encoding format described in Section 3.6.1 augmented with the optimization of incompressible segments described in Section 3.6.3. All additional design-dependent optimizations to the encoding format are explicitly mentioned for each design. In each design, all applicable performance optimizations of Section 3.7 are utilized.

The final data compressor designs come down to differences in factorization and additional encoding optimizations applicable to each factorization variant. The final designs and their descriptions are listed below.

- *greedy* - The baseline greedy SALZ variant described in Section 3.5.1.
- *greedy-opt* - Previous augmented with factor offset reuse and Golomb-Rice optimization described in Sections 3.6.2 and 3.6.4, respectively.
- *nongreedy* - The lazy non-greedy SALZ variant described in Section 3.5.2.
- *nongreedy-opt* - Previous augmented with factor offset reuse and Golomb-Rice optimization described in Sections 3.6.2 and 3.6.4, respectively.
- *mincost* - Standard minimum-cost SALZ variant described in Section 3.5.3.
- *mincost-opt* - Previous augmented with factor offset reuse described in Section 3.6.2.
- *dp-mincost* - Dynamic programming minimum-cost SALZ variant described in Section 3.5.4.

4.4 Benchmark results

In this section, I summarize the observations made from the SALZ benchmarks, which are available in Appendix A. The benchmarks measured compression ratio, and compression

and decompression times across all payloads and block sizes. The results are further grouped by their utilization of minimum-cost heuristics. I start by drawing observations from the decompression times, followed by compression times, and finally I discuss the compression ratios.

The decompression times are generally very close to each other. The decoding stages of *greedy*, *nongreedy*, *mincost* and *dp-mincost* designs are identical to each other. The *greedy-opt*, *nongreedy-opt* and *mincost* designs extend the baseline decoding stage with offset reuse. Additionally, *greedy-opt* and *nongreedy-opt* designs utilize a variable Golomb-Rice base. The fastest decompression times for non-minimum-cost designs are obtained with either *greedy* or *nongreedy* designs, depending on the payload and block size. The result is expected, as the decoding stages of the designs utilizing encoding optimizations contain additional complexity. Among the minimum-cost designs, the best results are generally obtained with *dp-mincost* design, which is surprising as the decoding stage and the produced compression ratios are identical to the *mincost* design. A closer look at the compressed output reveals that even though the encoding costs are identical, the factorizations are usually different due to the difference in the direction of the minimum-cost optimization. There are a couple of exceptions when the *mincost* design can match or slightly outperform the *dp-mincost* design regarding decompression time, but they are not significant enough to invalidate the general observation. Perhaps even more surprisingly, the *dp-mincost* design corresponds consistently to the fastest decompression times over all designs. It seems that the *dp-mincost* design can generally produce a factorization that exhibits a greater degree of locality in factor selection when compared to other designs. The only clear outlier is the *zero8* payload, for which the decompression times are almost the same between all designs. However, that is expected as the produced factorization is identical between all designs.

The most obvious observation about the compression times is that they increase proportional to the complexity of the data compressor design. In addition, the designs utilizing encoding enhancements are generally slower than their corresponding "vanilla" variants. From the non-minimum-cost designs, the *greedy* design is the fastest, followed by *greedy-opt*, *nongreedy* and *nongreedy-opt* designs. There are some occasions when *greedy-opt* and *nongreedy* can match or outperform *greedy*, and when *nongreedy* can beat *greedy-opt*, but they are not enough to disrupt the generally observed pattern. The *dp-mincost* design dominates the minimum-cost category, followed by *mincost* and *mincost-opt*. Again, there are some rare exceptions, namely *mincost* outperforming *dp-mincost* on some certain pay-

loads and block sizes. Over all designs, *greedy* is consistently the fastest one. The increase in overhead between non-minimum-cost and minimum-cost designs is quite large, and especially visible with larger block sizes with slightly less compressible payloads. In addition, with minimum-cost designs, the proportional increase in overhead reduces when the block size increases, which is the opposite of what happens with non-minimum-cost designs.

With the compression ratio, the benchmark results exhibit more variance than with compression and decompression times. Generally, the compression ratios are quite close to each other when designs are compared within their respective groups. When designs are compared across the groups, the difference in compression ratio is more dramatic. As expected, the most complex designs provide generally the best compression ratios. The best compression ratios are obtained with the *nongreedy-opt* design for the non-minimum-cost designs and the *mincost-opt* design for the minimum-cost design, even though on some occasions on small block sizes *mincost* and *dp-mincost* are able to match with it. Over all the designs, *mincost-opt* consistently produces the best results. The 32kiB block size is an exception to the previous as with many payloads, the best results are obtained with *nongreedy-opt* design. Another exception is the *proteins8* payload on small block sizes as even *greedy-opt* can beat *mincost-opt*. Like with the decompression times, the *zero8* payload is again an outlier as the produced factorization is identical between the compressor designs, such that the compression ratios are identical to each other. The exception to this are the *greedy-opt* and *nongreedy-opt* designs which achieve extremely high compression ratios due to their variable Golomb-Rice base optimization.

The best compressor design depends significantly on the objectives, desired behaviour, and application. When all factors are considered, there are three final designs that deserve to be highlighted. For the best compression, the clear winner is *mincost-opt*, which can consistently achieve the highest compression ratios by spending more time. On the other hand, almost as high compression ratios can be achieved with *dp-mincost*, which additionally minimizes the time spent on decompression. If the time spent on compression is a concern, *nongreedy-opt* seems to be the best choice, as it provides the best compression ratios among the non-minimum-cost designs, while not being significantly slower than the other designs of the same group, but being significantly faster than the minimum-cost designs. Since the decompression times are so close to each other, the choice of the most suitable data compressor design comes down to a trade-off between the compression ratio and compression time, as is usual with data compression.

5 Discussion

In the previous chapter, I described how SALZ was benchmarked and how different SALZ designs compare with each other. In this chapter, I introduce a group of selected third-party data compressors and compare them with SALZ. Then I continue by stating some starting points for future research, and finally I conclude this thesis with a section that provides a summary, reflection, and my final conclusions.

5.1 Comparison to third-party data compressors

In this section, I will provide a comparison of SALZ with selected third-party data compressors. Since I developed SALZ as a command-line interface (CLI) tool for Linux, I chose to include the de facto standard free and open-source CLI tools for Unix-like operating systems: `gzip` (GNU Gzip 1.10), `bzip2` (bzip2 1.0.8) and `xz` (XZ Utils 5.2.4). Additionally, I wanted to include some of the recent attempts at developing a high-speed data compressor. From these, I chose to include `lz4` (LZ4 1.9.2) and `zstd` (Zstandard 1.4.4). Also, I would have liked to include promising *Snappy*[1], but unfortunately it was not readily available as a CLI tool. All third-party tools were obtained as precompiled binaries from Ubuntu 20.04 package repositories, as the latest available versions. They were benchmarked with their minimum, maximum and default compression levels. However, only two compression levels were benchmarked for `lz4`, as its default and minimum levels are the same. For `zstd`, two additional compression levels were benchmarked, as the default range of compression levels can be extended by supplying additional command-line arguments. All benchmarks were performed using the SALZ benchmark procedure described in Section 4.2. The benchmark results of third-party data compressors for *silesia* payload are available in Appendix C.

`gzip`[2] was developed for the *GNU* project by Jean-Loup Gailly and Mark Adler and first released in 1992. Currently, it uses the *Deflate* algorithm, which was developed by Philip Katz in 1993 [22, Ch. 6.25]. The Deflate algorithm was specified as IETF RFC

[1] Snappy is a recent data compression library developed by Google. https://github.com/google/snappy (April 27, 2022)

[2] https://www.gnu.org/software/gzip/ (April 27, 2022)

1951[1] in 1996. The Deflate algorithm combines a variation of the LZ77 factorization with *Huffman coding*. In principle, the LZ77 factorization acts as a preprocessing stage for the Huffman coding. `gzip` has well-balanced performance. It provides good compression ratios, and it is neither fast nor slow in compression or decompression. Additionally, it is well-known and widely utilized, which is why I refer to it also when introducing other third-party data compressors. When low compression levels are compared to non-minimum-cost SALZ designs, we observe that the compression ratios are quite comparable, but `gzip` has lower compression times. However, with medium compression levels, compression times become comparable, but `gzip` produces higher compression ratios. When medium to high compression levels are compared with minimum-cost SALZ designs, we observe that SALZ can provide comparable or even slightly higher compression ratios with slightly lower or comparable compression times. In decompression, SALZ is always faster than `gzip`.

`bzip2`[2] was developed by Julian Seward and first released in 1996. `bzip2` uses a multi-stage algorithm that combines *Run-length encoding*, *Burrows-Wheeler transform*, *Move-to-front transform* and Huffman coding. Generally, `bzip2` provides higher compression ratios than `gzip`, but with higher compression and decompression times. When compared to SALZ, we observe that `bzip2` provides higher compression ratios than any SALZ design, with compression times that are comparable to only minimum-cost designs on highest block sizes. With decompression, SALZ is significantly faster than `bzip2`.

`xz`[3] was developed by Lasse Collin and first released in 2010. It uses the *LZMA* (Lempel-Ziv-Markov chain) algorithm, which was developed by Igor Pavlov in 1996 [22, Ch. 6.26]. The LZMA algorithm resembles the Deflate algorithm, but utilizes *Range coding* for post-processing instead of Huffman coding. The goal of `xz` is to provide extremely high compression ratios regardless of compression time, but without sacrificing fast decompression. Generally, `xz` provides significantly higher compression ratios than `gzip`, but it is extremely slow in compression. The single exception to the previous statement is `xz` at its lowest compression levels, where it has compression times comparable to `gzip`'s highest compression levels. The decompression times of `xz` are slightly higher than those of `gzip`. Compared to `bzip2`, `xz` is slower in compression, faster in decompression, and produces consistently higher compression ratios. The comparison between `xz` and SALZ leads to identical observations as in the comparison between `xz` and `gzip`.

[1] https://datatracker.ietf.org/doc/html/rfc1951 (April 27, 2022)
[2] https://www.sourceware.org/bzip2/ (April 27, 2022)
[3] https://tukaani.org/xz/ (April 27, 2022)

`lz4`[1] is a free and open-source data compressor developed by Yan Collet and first released in 2011. `lz4` is a variation of LZ77, that attempts to provide extremely fast compression and decompression, while providing reasonable compression ratios. Due to its high performance, LZ4 has been widely adopted, for example, for network protocols, (compressed) file systems, and databases. The lower compression levels of `lz4` produce compression ratios significantly lower than `gzip`, but compression and decompression are extremely fast. The highest compression levels of `lz4`, also known as LZ4HC, produce compression ratios comparable to `gzip` on its lowest compression levels, with significantly higher compression time, but without sacrificing decompression speed. Against SALZ, the comparison results are quite similar to those of `gzip`. The only exception is that LZ4HC can match the compression ratio of all SALZ designs with smaller block sizes at a cost of inferior compression times. Although SALZ is fast in decompression, `lz4` can outperform it by a wide margin.

`zstd`[2] is another free and open-source data compressor developed by Yan Collet. It was first released in 2015 and has been specified as IETF RFC 8878[3] in 2018. Like `xz`, `zstd` uses an algorithm that resembles the Deflate algorithm, with the exception that it utilizes *Finite State Entropy* (FSE) in combination with Huffman coding. At lower compression levels, `zstd` provides compression ratios comparable to `gzip`, while being significantly faster in compression. At the higher compression levels, the compression ratios are slightly higher than with `bzip2`, but slightly lower than with `xz`, while the compression times are comparable to `xz`. The decompression times of `zstd` are quite comparable to `lz4`. Compared to SALZ, `zstd` produces comparable compression ratios with significantly lower compression times. The decompression times of `zstd` are roughly comparable with `lz4` and therefore lower than with any SALZ design.

On the basis of the above observations, I conclude that SALZ is not the best choice for any of the possible compression scenarios. Note that because of the trade-off between compression ratio and time, not all imaginable scenarios are possible. If fast compression is prioritized at the cost of the compression ratio, `lz4` is the obvious choice. In the opposite scenario, when a high compression ratio is prioritized at the cost of compression time, `xz` seems to be the best choice. In case fast decompression times are prioritized, there are multiple possible choices depending on the secondary priority. If decompression time is the only priority, `lz4` is the obvious choice. For slightly better compression ratios, `zstd` is a great choice that results in only slightly inflated decompression times compared to

[1] https://lz4.github.io/lz4/ (April 27, 2022)
[2] https://facebook.github.io/zstd/ (April 27, 2022)
[3] https://datatracker.ietf.org/doc/html/rfc8878 (April 27, 2022)

`lz4`. For high compression ratios, `xz` provides the best decompression times, but it is not particularly fast. In general, `zstd` seems to be the current best choice for a general-purpose data compressor, as it can outperform `gzip` in every measured aspect.

5.2 Future work

Although this thesis comprehensively documents the practical aspects of designing and implementing a suffix array-based Lempel-Ziv data compressor, I have identified some aspects for which further study could prove beneficial.

Since in this thesis I focused on developing a general-purpose data compressor, I strived to keep both running times and memory consumption reasonable. In Section 3.3, I explained how the chosen block size influences the suffix array construction times. Additionally, the total memory consumption of all SALZ variants introduced in Section 4.3 is quite high, between $14n + O(1)$ and $22n + O(1)$ bytes, where n is the chosen block size. For these reasons, I developed SALZ mostly by relying on observations made from benchmarks executed on modest block sizes. Exploring different trade-offs, e.g., by experimenting with larger block sizes along with suitable encodings, could therefore provide a wider understanding of what is possible.

When I discussed the factorization and encoding in Section 3.5 and Section 3.6, respectively, I concluded that utilizing or emulating a sliding window algorithm is not feasible with suffix array-based methods. I reasoned that it has an adverse effect on the quality of extracted factors and that it also prevents efficient encoding of factor offsets. However, the significance of this limitation was not measured. Therefore, I think it could be beneficial to measure the difference in compression ratios between fixed window and sliding window based compression algorithms.

In Section 3.6.1, I explained how I committed to the LZSS-type encoding, as opposed to a token-based encoding, backed by observations of factor lengths. This commitment resulted in the inability to encode longer runs of literals efficiently. Furthermore, the chosen encoding type also caused challenges in factor offset reuse introduced in Section 3.6.2. As a result, I could only reuse the previous factor offset, which proved to be only slightly beneficial with respect to the compression ratio. I am convinced that exploring different token-based encodings could prove beneficial in the form of an increased compression ratio. For example, a token-based encoding could be designed around the idea of reusing a constant number of previous factors. The format could further be augmented by efficient

encoding of literal runs and encoding factor offsets using multiple switch controlled fixed lengths, instead of encoding them using a single variable-length scheme.

The last future research point is to further reduce the greediness in the suffix array-based Lempel-Ziv factorization. As concluded in Section 3.5.3, the factor candidates obtained from the PSV and NSV arrays do not necessarily result in factors that are optimal with respect to their encoded size. It is also clear that the suffix array and its derivatives contain a great amount of information that is currently not being used. Therefore, it could be beneficial to explore novel non-greedy ways to utilize the suffix array in Lempel-Ziv factorization. A relatively straightforward starting point could be to check some constant number of psv and nsv values for each factor starting position, hoping to find only slightly shorter factors with significantly smaller offsets.

5.3 Conclusions

In this thesis, I described the design and implementation of SALZ, a prototype of a suffix array-based Lempel-Ziv data compressor, from the ground up. I chose to approach the project with an experimental and exploratory approach. Therefore, I started by carefully curating a benchmark payload collection and setting up a benchmarking framework. I proceeded to collect a set of statistics to use as a guide for building an initial prototype to act as a starting point for the exploratory process. The statistics included times spent on data compression subtasks and information on the benchmark payloads. The statistics showed that the suffix array construction times dominated greatly over all other subtasks of data compression, which led me to suspect that my hypothesis would prove wrong. To provide meaningful measurements, I continued by designing a minimalistic file format, so that I could measure compression ratios and disk-to-disk compression and decompression times.

I started the exploratory phase by experimenting with suffix array construction. I confirmed that suffix array construction via ordinary string sorting was not a viable option, even for less-repetitive payloads. I came across a new and promising suffix array construction library, *libsais*, which I observed to consistently outperform *libdivsufsort*, the current default choice, by wide margins. During my thesis work, I also provided a minor contribution to *libsais*.[1] I then continued by experimenting with lcp-comparison, which resulted in the most important contribution of my thesis. Using the properties of the LPF array,

[1] https://github.com/IlyaGrebnov/libsais/issues/3 (May 3, 2022)

I could reduce the number of symbol comparisons performed during the Lempel-Ziv factorization. By performing the comparison using multi-byte words and efficient compiler intrinsics, I could compare the remaining symbols faster. I observed this technique to provide consistently over 80% improvement over the baseline solution. When exploring the LZ77 factorization, unfortunately, I observed that the traditional greedy approach did not result in compression ratios that would be justifiable considering the compression times. I therefore decided to use even more time and pursue higher compression ratios by reducing the amount of greediness and employing cost optimization heuristics. In this way, I was able to achieve significantly higher compression ratios with a justifiable increase in compression time. The second most important contribution of my thesis is the dynamic minimum-cost factorization algorithm, which is the fastest of the high-compression variants due to optimized memory access patterns and reduced number of memory accesses. The exploratory phase was concluded with experiments on encoding. The final encoding proved to be not only efficient, but also fast. Most surprisingly, even though the encoding is mostly bit-oriented, the decompression times are close to those of the fastest well-known data compressors.

Before continuing to final benchmarks and evaluation, I tried my best to optimize all the techniques using both well-known general-purpose optimizations and domain-specific optimizations. Finally, I combined the techniques I had developed during the exploratory phase and built seven different data compressor designs. Evaluation of the final benchmarks confirmed that my hypothesis was wrong. Even if the final prototypes compress well, without being particularly slow, they are too slow for the compression ratios they provide to be competitive. The main bottleneck is the use of a suffix array, as its construction constitutes the majority of the compression time. Additionally, the suffix array also limits the compression ratio, as I observed that the highest compression ratios do not necessarily result from utilizing the longest previous factors. However, it should be noted that the compression ratios themselves are very competitive when SALZ is compared to other data compressors that rely exclusively on dictionary compression, i.e., refrain from using an additional entropy encoding stage.

Despite the fact that my hypothesis proved to be wrong, I consider all the objectives set for this thesis to have been met. To summarize, I found out that suffix array construction takes currently too much time to achieve fast data compression, and that it is not possible to achieve the higher compression ratios using it, or relying exclusively on dictionary compression. However, the suffix array was observed to provide factors of consistent

quality, which proved beneficial when fitting an encoding scheme to the chosen factorization. I was further able to augment greedy LZ77 factorization by employing non-greedy techniques and cost optimization heuristics. Additionally, I found various practicalities that are necessary for designing and implementing a general-purpose data compressor. Although I outlined some points of interest for further research in the previous section, I believe that the outcome of this thesis will remain correct. However, significant further improvements in suffix array construction or in the update cost of dynamic suffix arrays could show otherwise. Nevertheless, suffix array-based Lempel-Ziv factorization (or data compression) remains useful for scenarios where the suffix array is anyway needed for some other purpose.

Bibliography

[1] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. "Efficient Index Compression in DB2 LUW". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1462–1473. ISSN: 2150-8097. DOI: 10.14778/1687553.1687573.

[2] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. "The smallest grammar problem". In: *IEEE Transactions on Information Theory* 51.7 (2005), pp. 2554–2576. DOI: 10.1109/TIT.2005.850116.

[3] M. Crochemore and L. Ilie. "Computing Longest Previous Factor in Linear Time and Applications". In: *Information Processing Letters* 106.2 (Apr. 2008), pp. 75–80. ISSN: 0020-0190. DOI: 10.1016/J.IPL.2007.10.006.

[4] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. "Linear-Time Computation of Local Periods". In: *Mathematical Foundations of Computer Science 2003*. Ed. by B. Rovan and P. Vojtáš. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 388–397. ISBN: 978-3-540-45138-9.

[5] P. Ferragina, I. Nitto, and R. Venturini. "On the Bit-Complexity of Lempel–Ziv Compression". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1521–1541. DOI: 10.1137/120869511.

[6] S. Golomb. "Run-length encodings (Corresp.)" In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401. DOI: 10.1109/TIT.1966.1053907.

[7] K. Goto and H. Bannai. "Simpler and Faster Lempel Ziv Factorization". In: *2013 Data Compression Conference*. 2013, pp. 133–142. DOI: 10.1109/DCC.2013.21.

[8] R. N. Horspool. "The Effect of Non-Greedy Parsing in Ziv-Lempel Compression Methods". In: *Proceedings DCC '95 Data Compression Conference*. IEEE, 1995, pp. 302–311. ISBN: 0818670126.

[9] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. "Lazy Lempel-Ziv Factorization Algorithms". In: *ACM J. Exp. Algorithmics* 21 (Oct. 2016). ISSN: 1084-6654. DOI: 10.1145/2699876.

[10] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. "Linear Time Lempel-Ziv Factorization: Simple, Fast, Small". In: *Combinatorial Pattern Matching*. Ed. by J. Fischer and P. Sanders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 189–200. ISBN: 978-3-642-38905-4.

[11] D. Kempa and S. J. Puglisi. "Lempel-Ziv Factorization: Simple, Fast, Practical". In: *2013 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pp. 103–112. DOI: 10.1137/1.9781611972931.9.

[12] R. Kolpakov and G. Kucherov. "Finding maximal repetitions in a word in linear time". In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 596–604. DOI: 10.1109/SFFCS.1999.814634.

[13] D. Lemire, N. Kurz, and C. Rupp. "Stream VByte: Faster byte-oriented integer compression". In: *Information Processing Letters* 130 (2018), pp. 1–6. ISSN: 0020-0190. DOI: 10.1016/J.IPL.2017.09.011.

[14] A. Lempel and J. Ziv. "On the Complexity of Finite Sequences". In: *IEEE Transactions on Information Theory* 22.1 (1976), pp. 75–81. DOI: 10.1109/TIT.1976.1055501.

[15] U. Manber and G. Myers. "Suffix Arrays: A New Method for On-Line String Searches". In: *SIAM Journal on Computing* 22.5 (1993), pp. 935–948. DOI: 10.1137/0222058.

[16] G. Navarro. "Indexing Highly Repetitive Collections". In: *Combinatorial Algorithms*. Ed. by S. Arumugam and W. F. Smyth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 274–279. ISBN: 978-3-642-35926-2.

[17] G. Navarro and V. Mäkinen. "Compressed Full-Text Indexes". In: *ACM Comput. Surv.* 39.1 (Apr. 2007), 2–es. ISSN: 0360-0300. DOI: 10.1145/1216370.1216372.

[18] G. Nong, S. Zhang, and W. H. Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: *2009 Data Compression Conference*. 2009, pp. 193–202. DOI: 10.1109/DCC.2009.42.

[19] E. Ohlebusch and S. Gog. "Lempel-Ziv Factorization Revisited". In: CPM'11. Palermo, Italy: Springer-Verlag, 2011, pp. 15–26. ISBN: 9783642214578.

[20] J. Plaisance, N. Kurz, and D. Lemire. "Vectorized VByte Decoding". In: *International Symposium on Web AlGorithms*. 2015.

[21] R. F. Rice. "Practical Universal Noiseless Coding". In: *Applications of Digital Image Processing III*. Ed. by A. G. Tescher. Vol. 0207. International Society for Optics and Photonics. SPIE, 1979, pp. 247–267. DOI: 10.1117/12.958253.

[22] D. Salomon. *Handbook of data compression*. 5th ed. London: Springer, 2010. ISBN: 1848829035.

[23] D. Salomon. *Variable-length codes for data compression*. London: Springer, 2007. ISBN: 1846289599.

[24] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. "Compression of Inverted Indexes For Fast Query Evaluation". In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '02. Tampere, Finland: Association for Computing Machinery, 2002, pp. 222–229. ISBN: 1581135610. DOI: 10.1145/564376.564416.

[25] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. 11th ed. New York, NY: Pearson, 2019. ISBN: 9780135160930.

[26] J. A. Storer and T. G. Szymanski. "Data Compression via Textual Substitution". In: *J. ACM* 29.4 (Oct. 1982), pp. 928–951. ISSN: 0004-5411. DOI: 10.1145/322344.322346.

[27] *The Complete MIDI 1.0 Detailed Specification*. Los Angeles, CA: The MIDI Manufacturers Association, 1996.

[28] S. Vigna. "Quasi-Succinct Indices". In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 83–92. ISBN: 9781450318693. DOI: 10.1145/2433396.2433409.

[29] H. E. Williams and J. Zobel. "Compressing Integers for Fast File Access". In: *The Computer Journal* 42.3 (Jan. 1999), pp. 193–201. ISSN: 0010-4620. DOI: 10.1093/COMJNL/42.3.193.

[30] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714.

Appendix A SALZ benchmarks

Table A.1: Benchmark with *dblp.xml8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	4.205	4.208	4.320	4.338	4.385	4.412	4.385
64 k	4.575	4.578	4.710	4.729	4.806	4.833	4.806
128 k	4.940	4.944	5.095	5.115	5.219	5.244	5.219
256 k	5.274	5.279	5.449	5.477	5.603	5.626	5.603
512 k	5.539	5.549	5.737	5.777	5.923	5.943	5.923
1 M	5.778	5.797	5.996	6.049	6.207	6.226	6.207
2 M	5.969	6.001	6.202	6.270	6.440	6.458	6.440
4 M	6.113	6.158	6.349	6.431	6.616	6.634	6.616
8 M	6.264	6.325	6.501	6.600	6.794	6.812	6.794
16 M	6.414	6.492	6.652	6.769	6.973	6.993	6.973
32 M	6.530	6.624	6.772	6.907	7.124	7.146	7.124
64 M	6.639	6.746	6.886	7.035	7.248	7.272	7.248
128 M	6.793	6.918	7.047	7.217	7.425	7.451	7.425
	Compression time (seconds)						
32 k	2.951	3.365	3.394	3.431	4.683	4.703	4.297
64 k	2.905	3.285	3.322	3.344	4.666	4.697	4.263
128 k	2.907	3.260	3.306	3.311	4.670	4.714	4.269
256 k	3.022	3.351	3.415	3.402	4.842	4.867	4.412
512 k	3.074	3.413	3.463	3.463	5.120	5.125	4.652
1 M	3.204	3.540	3.564	3.593	5.293	5.311	4.926
2 M	3.461	3.762	3.770	3.808	5.568	5.593	5.200
4 M	3.830	4.024	4.045	4.077	5.967	6.003	5.515
8 M	4.407	4.517	4.529	4.560	6.478	6.532	6.024
16 M	5.154	5.262	5.248	5.302	7.386	7.452	6.921
32 M	5.804	5.874	5.906	5.944	8.427	8.486	7.970
64 M	6.165	6.211	6.242	6.275	9.047	9.134	8.587
128 M	6.498	6.546	6.591	6.592	9.740	9.808	9.279
	Decompression time (seconds)						
32 k	0.186	0.216	0.190	0.208	0.193	0.196	0.186
64 k	0.178	0.206	0.179	0.199	0.183	0.188	0.177
128 k	0.168	0.194	0.172	0.188	0.173	0.178	0.167
256 k	0.160	0.187	0.163	0.180	0.164	0.169	0.158
512 k	0.158	0.182	0.159	0.177	0.159	0.165	0.155
1 M	0.162	0.184	0.162	0.177	0.160	0.168	0.156
2 M	0.166	0.187	0.166	0.180	0.163	0.169	0.160
4 M	0.166	0.187	0.166	0.180	0.161	0.167	0.159
8 M	0.166	0.190	0.167	0.182	0.163	0.168	0.159
16 M	0.183	0.202	0.184	0.198	0.174	0.182	0.175
32 M	0.228	0.263	0.242	0.253	0.219	0.224	0.224
64 M	0.252	0.295	0.287	0.278	0.244	0.261	0.245
128 M	0.282	0.343	0.313	0.326	0.273	0.290	0.271

Table A.2: Benchmark with *dna8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.804	2.852	2.820	2.858	2.948	2.948	2.948
64 k	2.849	2.876	2.863	2.884	3.008	3.010	3.008
128 k	2.910	2.923	2.923	2.934	3.068	3.071	3.068
256 k	2.960	2.965	2.971	2.976	3.121	3.125	3.121
512 k	2.971	2.972	2.982	2.985	3.141	3.145	3.141
1 M	3.004	3.004	3.015	3.018	3.167	3.172	3.167
2 M	3.030	3.030	3.040	3.044	3.193	3.198	3.193
4 M	3.027	3.026	3.039	3.042	3.196	3.201	3.196
8 M	3.048	3.048	3.064	3.066	3.213	3.216	3.213
16 M	3.071	3.071	3.088	3.091	3.234	3.238	3.234
32 M	3.077	3.077	3.097	3.099	3.244	3.247	3.244
64 M	3.096	3.096	3.116	3.118	3.258	3.261	3.258
128 M	3.128	3.128	3.149	3.152	3.284	3.286	3.284
	Compression time (seconds)						
32 k	4.271	4.545	4.596	4.703	6.698	6.713	5.687
64 k	4.268	4.496	4.590	4.666	6.833	6.857	5.823
128 k	4.270	4.451	4.560	4.619	6.770	6.804	5.846
256 k	4.353	4.517	4.611	4.663	6.943	6.960	6.047
512 k	4.388	4.570	4.658	4.692	7.295	7.252	6.386
1 M	4.445	4.636	4.708	4.728	7.441	7.397	6.760
2 M	4.636	4.785	4.849	4.893	7.678	7.639	7.111
4 M	4.939	5.039	5.087	5.148	8.052	8.057	7.551
8 M	5.412	5.489	5.568	5.583	8.758	8.736	8.115
16 M	6.099	6.239	6.292	6.384	11.141	11.172	10.515
32 M	6.739	6.875	6.989	6.998	14.121	14.201	13.575
64 M	7.294	7.227	7.326	7.320	15.861	15.916	15.430
128 M	7.590	7.661	7.643	7.723	17.750	17.916	17.336
	Decompression time (seconds)						
32 k	0.248	0.289	0.255	0.294	0.257	0.272	0.246
64 k	0.254	0.293	0.259	0.298	0.260	0.277	0.249
128 k	0.249	0.287	0.250	0.289	0.256	0.274	0.247
256 k	0.242	0.280	0.241	0.278	0.245	0.262	0.235
512 k	0.250	0.284	0.249	0.285	0.251	0.268	0.239
1 M	0.248	0.278	0.248	0.281	0.253	0.271	0.249
2 M	0.240	0.268	0.239	0.273	0.242	0.258	0.236
4 M	0.236	0.268	0.239	0.270	0.239	0.258	0.229
8 M	0.244	0.277	0.250	0.281	0.251	0.266	0.244
16 M	0.366	0.426	0.395	0.417	0.374	0.393	0.367
32 M	0.487	0.570	0.526	0.558	0.499	0.521	0.493
64 M	0.563	0.638	0.587	0.635	0.559	0.584	0.561
128 M	0.685	0.734	0.672	0.723	0.627	0.664	0.632

Table A.3: Benchmark with *english8* payload The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	1.974	2.068	2.035	2.111	2.091	2.091	2.091
64 k	2.084	2.162	2.151	2.210	2.221	2.221	2.221
128 k	2.197	2.257	2.268	2.315	2.347	2.347	2.347
256 k	2.299	2.346	2.375	2.412	2.465	2.465	2.465
512 k	2.380	2.418	2.461	2.490	2.564	2.565	2.564
1 M	2.510	2.539	2.596	2.617	2.710	2.711	2.710
2 M	2.710	2.728	2.803	2.818	2.931	2.936	2.931
4 M	2.823	2.832	2.918	2.927	3.060	3.068	3.060
8 M	2.952	2.955	3.048	3.054	3.200	3.207	3.200
16 M	3.052	3.054	3.149	3.156	3.312	3.320	3.312
32 M	3.169	3.171	3.268	3.276	3.446	3.455	3.446
64 M	3.208	3.210	3.307	3.315	3.488	3.498	3.488
128 M	3.245	3.246	3.343	3.350	3.526	3.535	3.526
	Compression time (seconds)						
32 k	4.315	4.600	4.575	4.765	6.457	6.473	5.748
64 k	4.072	4.345	4.356	4.532	6.363	6.355	5.588
128 k	3.919	4.186	4.195	4.356	6.174	6.209	5.433
256 k	3.980	4.217	4.259	4.385	6.321	6.327	5.591
512 k	4.022	4.266	4.307	4.431	6.616	6.579	5.879
1 M	4.130	4.346	4.393	4.498	6.759	6.722	6.172
2 M	4.320	4.566	4.564	4.666	6.935	6.930	6.431
4 M	4.652	4.945	4.852	4.943	7.331	7.328	6.830
8 M	5.223	5.415	5.400	5.468	7.988	7.978	7.404
16 M	6.055	6.259	6.218	6.320	9.690	9.726	9.092
32 M	6.842	6.978	7.037	7.109	11.770	11.819	11.206
64 M	7.298	7.414	7.464	7.493	13.076	13.159	12.650
128 M	7.831	7.935	8.003	7.961	14.485	14.541	14.178
	Decompression time (seconds)						
32 k	0.332	0.376	0.331	0.372	0.347	0.357	0.331
64 k	0.323	0.370	0.322	0.367	0.336	0.349	0.319
128 k	0.308	0.353	0.307	0.351	0.318	0.333	0.303
256 k	0.296	0.340	0.296	0.338	0.304	0.322	0.291
512 k	0.314	0.357	0.313	0.357	0.317	0.334	0.305
1 M	0.310	0.350	0.307	0.347	0.311	0.328	0.305
2 M	0.289	0.328	0.286	0.324	0.287	0.301	0.277
4 M	0.286	0.324	0.278	0.315	0.278	0.290	0.268
8 M	0.285	0.319	0.279	0.312	0.276	0.288	0.267
16 M	0.378	0.358	0.360	0.383	0.338	0.354	0.337
32 M	0.487	0.474	0.465	0.499	0.433	0.454	0.429
64 M	0.545	0.530	0.514	0.556	0.481	0.510	0.482
128 M	0.625	0.599	0.591	0.641	0.541	0.588	0.549

Table A.4: Benchmark with *enwik8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.127	2.199	2.181	2.239	2.230	2.232	2.230
64 k	2.233	2.293	2.294	2.342	2.356	2.358	2.356
128 k	2.334	2.383	2.401	2.444	2.473	2.476	2.473
256 k	2.419	2.462	2.493	2.529	2.578	2.581	2.578
512 k	2.478	2.514	2.559	2.588	2.660	2.662	2.660
1 M	2.542	2.571	2.630	2.652	2.740	2.743	2.740
2 M	2.605	2.626	2.696	2.710	2.817	2.820	2.817
4 M	2.650	2.664	2.742	2.748	2.875	2.878	2.875
8 M	2.709	2.714	2.800	2.801	2.940	2.943	2.940
16 M	2.769	2.770	2.860	2.862	3.010	3.013	3.010
32 M	2.811	2.812	2.903	2.905	3.064	3.068	3.064
64 M	2.852	2.853	2.945	2.947	3.110	3.113	3.110
128 M	2.916	2.917	3.010	3.011	3.180	3.183	3.180
	Compression time (seconds)						
32 k	4.360	4.766	4.646	4.858	6.323	6.318	5.719
64 k	4.272	4.642	4.540	4.723	6.457	6.375	5.685
128 k	4.144	4.506	4.417	4.582	6.336	6.281	5.582
256 k	3.983	4.327	4.257	4.421	6.261	6.212	5.546
512 k	4.011	4.351	4.301	4.430	6.515	6.453	5.773
1 M	4.085	4.386	4.383	4.484	6.669	6.588	6.036
2 M	4.279	4.529	4.531	4.634	6.847	6.799	6.297
4 M	4.565	4.812	4.790	4.879	7.350	7.173	6.663
8 M	5.083	5.308	5.254	5.358	7.772	7.764	7.506
16 M	5.952	6.166	6.123	6.230	9.445	9.464	8.838
32 M	6.730	6.889	6.918	7.010	11.432	11.492	10.810
64 M	7.177	7.323	7.336	7.410	12.600	12.687	12.022
128 M	7.688	7.837	7.812	7.870	13.785	13.844	13.364
	Decompression time (seconds)						
32 k	0.325	0.375	0.325	0.367	0.338	0.347	0.323
64 k	0.319	0.368	0.317	0.360	0.330	0.338	0.314
128 k	0.307	0.358	0.306	0.349	0.317	0.328	0.302
256 k	0.298	0.346	0.296	0.341	0.306	0.318	0.292
512 k	0.318	0.363	0.314	0.356	0.319	0.332	0.308
1 M	0.318	0.362	0.315	0.354	0.321	0.333	0.311
2 M	0.309	0.356	0.308	0.346	0.310	0.321	0.298
4 M	0.308	0.349	0.305	0.338	0.298	0.313	0.289
8 M	0.311	0.351	0.305	0.342	0.304	0.316	0.322
16 M	0.427	0.407	0.408	0.431	0.391	0.404	0.383
32 M	0.553	0.530	0.526	0.566	0.496	0.516	0.488
64 M	0.611	0.587	0.588	0.627	0.548	0.574	0.543
128 M	0.678	0.663	0.648	0.706	0.604	0.646	0.601

Table A.5: Benchmark with *kernel8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	3.745	3.782	3.847	3.901	3.908	3.945	3.908
64 k	4.056	4.091	4.180	4.236	4.264	4.305	4.264
128 k	4.344	4.376	4.487	4.544	4.591	4.635	4.591
256 k	4.590	4.621	4.752	4.811	4.881	4.926	4.881
512 k	4.766	4.795	4.949	5.009	5.103	5.150	5.103
1 M	4.921	4.949	5.122	5.185	5.297	5.345	5.297
2 M	5.039	5.063	5.253	5.316	5.450	5.498	5.450
4 M	5.106	5.131	5.326	5.390	5.546	5.595	5.546
8 M	5.168	5.193	5.393	5.460	5.631	5.680	5.631
16 M	5.218	5.239	5.449	5.510	5.704	5.753	5.704
32 M	5.239	5.259	5.476	5.541	5.758	5.807	5.758
64 M	5.279	5.292	5.522	5.586	5.816	5.865	5.816
128 M	5.341	5.362	5.592	5.664	5.903	5.951	5.903
	Compression time (seconds)						
32 k	3.382	3.726	3.597	3.672	5.096	5.147	4.557
64 k	3.298	3.560	3.508	3.587	5.042	5.064	4.515
128 k	3.260	3.488	3.461	3.530	5.007	5.016	4.483
256 k	3.305	3.528	3.517	3.566	5.132	5.125	4.600
512 k	3.326	3.539	3.534	3.574	5.347	5.335	4.774
1 M	3.401	3.606	3.590	3.640	5.505	5.482	4.962
2 M	3.553	3.741	3.723	3.778	5.662	5.652	5.213
4 M	3.785	3.937	3.936	3.974	5.975	5.969	5.546
8 M	4.145	4.280	4.277	4.328	6.355	6.352	5.861
16 M	4.683	4.827	4.812	4.869	7.037	7.068	6.519
32 M	5.207	5.362	5.348	5.384	7.901	7.939	7.420
64 M	5.549	5.677	5.679	5.723	8.502	8.539	8.075
128 M	5.914	6.068	6.059	6.134	9.236	9.282	8.798
	Decompression time (seconds)						
32 k	0.214	0.248	0.213	0.233	0.221	0.228	0.209
64 k	0.205	0.235	0.203	0.225	0.208	0.216	0.199
128 k	0.195	0.225	0.193	0.216	0.197	0.205	0.189
256 k	0.188	0.215	0.186	0.208	0.189	0.196	0.182
512 k	0.188	0.212	0.183	0.205	0.185	0.193	0.179
1 M	0.191	0.215	0.186	0.208	0.188	0.196	0.184
2 M	0.191	0.215	0.187	0.207	0.187	0.194	0.183
4 M	0.192	0.215	0.187	0.208	0.186	0.193	0.183
8 M	0.198	0.219	0.191	0.212	0.188	0.194	0.184
16 M	0.207	0.227	0.200	0.221	0.196	0.206	0.194
32 M	0.261	0.280	0.250	0.270	0.233	0.244	0.234
64 M	0.294	0.315	0.276	0.299	0.260	0.273	0.257
128 M	0.332	0.350	0.307	0.329	0.287	0.300	0.285

Table A.6: Benchmark with *pitches* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.524	2.586	2.558	2.614	2.602	2.612	2.602
64 k	2.558	2.612	2.594	2.644	2.650	2.660	2.650
128 k	2.581	2.629	2.619	2.663	2.688	2.697	2.688
256 k	2.593	2.635	2.634	2.672	2.716	2.725	2.716
512 k	2.584	2.621	2.628	2.662	2.725	2.733	2.725
1 M	2.575	2.609	2.623	2.654	2.733	2.742	2.733
2 M	2.572	2.602	2.623	2.650	2.745	2.755	2.745
4 M	2.559	2.583	2.610	2.633	2.742	2.752	2.742
8 M	2.554	2.575	2.605	2.624	2.747	2.756	2.747
16 M	2.550	2.564	2.602	2.614	2.753	2.762	2.753
32 M	2.555	2.565	2.608	2.617	2.770	2.780	2.770
64 M	2.624	2.629	2.681	2.686	2.857	2.868	2.857
128 M	2.624	2.629	2.681	2.686	2.857	2.868	2.857
	Compression time (seconds)						
32 k	2.506	2.687	2.647	2.733	3.330	3.296	3.042
64 k	2.552	2.679	2.639	2.729	3.409	3.375	3.113
128 k	2.571	2.673	2.644	2.726	3.476	3.445	3.153
256 k	2.644	2.707	2.679	2.763	3.604	3.567	3.244
512 k	2.675	2.723	2.707	2.774	3.754	3.709	3.343
1 M	2.692	2.746	2.708	2.798	3.849	3.804	3.431
2 M	2.761	2.778	2.753	2.835	3.914	3.873	3.540
4 M	2.772	2.862	2.839	2.924	4.039	4.013	3.702
8 M	2.914	2.971	2.929	2.989	4.182	4.164	3.839
16 M	3.117	3.281	3.213	3.293	4.658	4.659	4.327
32 M	3.419	3.650	3.545	3.603	5.347	5.362	5.073
64 M	3.726	4.041	3.892	3.958	6.183	6.235	5.929
128 M	3.718	4.118	3.886	4.049	6.227	6.262	5.919
	Decompression time (seconds)						
32 k	0.151	0.170	0.153	0.167	0.152	0.155	0.144
64 k	0.153	0.168	0.151	0.166	0.156	0.159	0.147
128 k	0.155	0.169	0.151	0.167	0.157	0.162	0.149
256 k	0.158	0.171	0.152	0.170	0.158	0.164	0.149
512 k	0.169	0.183	0.162	0.180	0.167	0.174	0.160
1 M	0.175	0.188	0.169	0.187	0.174	0.181	0.167
2 M	0.176	0.186	0.165	0.185	0.171	0.176	0.165
4 M	0.173	0.188	0.167	0.187	0.171	0.178	0.165
8 M	0.175	0.191	0.171	0.189	0.177	0.182	0.169
16 M	0.189	0.204	0.187	0.203	0.191	0.195	0.187
32 M	0.224	0.257	0.244	0.254	0.230	0.238	0.226
64 M	0.268	0.313	0.289	0.300	0.265	0.275	0.262
128 M	0.267	0.310	0.298	0.308	0.268	0.270	0.268

Table A.7: Benchmark with *proteins8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	1.417	1.572	1.425	1.579	1.476	1.477	1.476
64 k	1.448	1.589	1.457	1.597	1.514	1.515	1.514
128 k	1.481	1.604	1.491	1.611	1.550	1.552	1.550
256 k	1.508	1.614	1.518	1.623	1.584	1.586	1.584
512 k	1.522	1.614	1.534	1.625	1.612	1.615	1.612
1 M	1.543	1.624	1.558	1.639	1.643	1.646	1.643
2 M	1.573	1.643	1.591	1.662	1.678	1.683	1.678
4 M	1.633	1.690	1.653	1.713	1.743	1.750	1.743
8 M	1.706	1.754	1.730	1.782	1.821	1.832	1.821
16 M	1.781	1.822	1.812	1.860	1.908	1.925	1.908
32 M	1.826	1.862	1.862	1.910	1.969	1.992	1.969
64 M	1.846	1.880	1.885	1.930	1.995	2.019	1.995
128 M	1.943	1.943	1.993	2.037	2.108	2.141	2.108
	Compression time (seconds)						
32 k	5.152	5.327	5.340	5.551	6.972	6.976	6.442
64 k	5.207	5.384	5.430	5.623	7.120	7.090	6.579
128 k	5.209	5.356	5.396	5.593	7.143	7.117	6.574
256 k	5.250	5.413	5.449	5.630	7.362	7.319	6.762
512 k	5.284	5.462	5.516	5.670	7.684	7.625	7.064
1 M	5.315	5.542	5.572	5.750	7.828	7.760	7.362
2 M	5.337	5.562	5.586	5.763	7.995	7.942	7.635
4 M	5.796	5.912	5.939	6.103	8.382	8.375	8.061
8 M	6.039	6.168	6.203	6.358	8.884	8.887	8.520
16 M	6.994	7.189	7.270	7.437	11.101	11.143	10.746
32 M	7.883	8.142	8.274	8.410	14.150	14.239	14.049
64 M	8.384	8.627	8.839	9.004	16.114	16.195	16.058
128 M	8.948	8.948	9.357	9.541	17.989	18.058	18.188
	Decompression time (seconds)						
32 k	0.425	0.432	0.415	0.423	0.450	0.466	0.420
64 k	0.426	0.440	0.418	0.430	0.440	0.457	0.411
128 k	0.420	0.439	0.409	0.431	0.426	0.444	0.399
256 k	0.433	0.448	0.422	0.445	0.428	0.447	0.405
512 k	0.448	0.479	0.441	0.481	0.445	0.472	0.427
1 M	0.424	0.465	0.409	0.473	0.436	0.456	0.418
2 M	0.396	0.442	0.385	0.449	0.416	0.438	0.395
4 M	0.400	0.443	0.395	0.446	0.412	0.434	0.393
8 M	0.402	0.440	0.395	0.440	0.403	0.429	0.388
16 M	0.514	0.587	0.537	0.569	0.502	0.526	0.495
32 M	0.681	0.800	0.732	0.773	0.645	0.675	0.639
64 M	0.783	0.930	0.845	0.899	0.732	0.774	0.725
128 M	0.855	0.855	0.932	0.985	0.812	0.850	0.795

Table A.8: Benchmark with *silesia* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.451	2.524	2.508	2.581	2.551	2.573	2.551
64 k	2.561	2.630	2.625	2.693	2.681	2.704	2.681
128 k	2.656	2.717	2.725	2.787	2.793	2.816	2.793
256 k	2.735	2.790	2.810	2.866	2.890	2.913	2.890
512 k	2.789	2.838	2.871	2.923	2.966	2.989	2.966
1 M	2.840	2.885	2.930	2.979	3.036	3.059	3.036
2 M	2.888	2.931	2.985	3.032	3.102	3.124	3.102
4 M	2.927	2.963	3.026	3.067	3.150	3.171	3.150
8 M	2.965	2.993	3.065	3.098	3.193	3.214	3.193
16 M	2.996	3.019	3.096	3.126	3.228	3.248	3.228
32 M	3.007	3.029	3.107	3.136	3.245	3.265	3.245
64 M	3.018	3.035	3.119	3.144	3.262	3.281	3.262
128 M	3.019	3.025	3.119	3.133	3.262	3.281	3.262
	Compression time (seconds)						
32 k	7.508	8.375	8.435	8.842	11.568	11.729	10.582
64 k	7.293	8.120	8.207	8.548	11.374	11.605	10.427
128 k	7.244	8.049	8.132	8.457	11.329	11.601	10.391
256 k	7.487	8.208	8.290	8.589	11.638	11.895	10.688
512 k	7.544	8.258	8.355	8.639	12.162	12.359	11.144
1 M	7.680	8.454	8.502	8.822	12.547	12.602	11.737
2 M	8.119	8.817	8.878	9.190	12.934	12.983	12.232
4 M	8.813	9.331	9.413	9.716	13.623	13.666	12.944
8 M	9.726	10.087	10.161	10.395	14.464	14.557	13.693
16 M	10.763	11.207	11.220	11.439	15.861	15.970	15.052
32 M	11.744	12.185	12.167	12.436	17.421	17.546	16.657
64 M	12.776	13.052	13.050	13.215	18.539	19.012	18.142
128 M	13.315	13.840	13.775	13.978	19.544	20.078	19.226
	Decompression time (seconds)						
32 k	0.516	0.600	0.536	0.596	0.561	0.579	0.536
64 k	0.506	0.583	0.520	0.582	0.543	0.567	0.522
128 k	0.489	0.569	0.503	0.563	0.524	0.552	0.505
256 k	0.492	0.569	0.502	0.563	0.522	0.547	0.502
512 k	0.525	0.587	0.522	0.582	0.538	0.564	0.522
1 M	0.531	0.588	0.525	0.584	0.543	0.562	0.528
2 M	0.521	0.582	0.514	0.575	0.527	0.546	0.514
4 M	0.518	0.581	0.514	0.573	0.523	0.543	0.509
8 M	0.525	0.591	0.524	0.581	0.527	0.549	0.517
16 M	0.550	0.616	0.554	0.604	0.547	0.563	0.535
32 M	0.605	0.684	0.611	0.661	0.579	0.617	0.583
64 M	0.715	0.785	0.690	0.739	0.619	0.678	0.647
128 M	0.746	0.802	0.712	0.762	0.638	0.691	0.664

Table A.9: Benchmark with *silesia8h* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.740	2.828	2.816	2.908	2.861	2.906	2.861
64 k	2.835	2.915	2.918	3.003	2.974	3.020	2.974
128 k	2.917	2.989	3.008	3.086	3.073	3.119	3.073
256 k	2.987	3.052	3.087	3.159	3.162	3.207	3.162
512 k	3.034	3.093	3.144	3.211	3.233	3.277	3.233
1 M	3.073	3.127	3.196	3.258	3.296	3.339	3.296
2 M	3.117	3.167	3.250	3.308	3.363	3.404	3.363
4 M	3.158	3.201	3.295	3.349	3.416	3.456	3.416
8 M	3.190	3.226	3.330	3.377	3.460	3.498	3.460
16 M	3.212	3.242	3.354	3.398	3.492	3.528	3.492
32 M	3.224	3.250	3.367	3.407	3.512	3.547	3.512
64 M	3.225	3.249	3.372	3.412	3.524	3.558	3.524
128 M	3.233	3.246	3.380	3.403	3.530	3.564	3.530
	Compression time (seconds)						
32 k	3.549	3.630	3.666	3.748	4.997	4.821	4.594
64 k	3.536	3.624	3.648	3.735	5.049	4.952	4.604
128 k	3.523	3.618	3.635	3.727	5.067	4.985	4.600
256 k	3.596	3.707	3.723	3.810	5.204	5.154	4.748
512 k	3.628	3.747	3.749	3.847	5.438	5.362	4.954
1 M	3.677	3.832	3.818	3.914	5.610	5.527	5.191
2 M	3.803	3.968	3.957	4.072	5.795	5.702	5.419
4 M	4.067	4.182	4.185	4.300	6.111	6.066	5.776
8 M	4.462	4.538	4.548	4.631	6.485	6.450	6.128
16 M	4.938	5.052	5.034	5.127	7.105	7.091	6.715
32 M	5.393	5.534	5.517	5.601	7.782	7.800	7.440
64 M	5.820	5.997	5.947	6.042	8.582	8.625	8.235
128 M	6.019	6.239	6.164	6.277	8.852	8.890	8.539
	Decompression time (seconds)						
32 k	0.239	0.259	0.227	0.251	0.234	0.234	0.225
64 k	0.236	0.257	0.223	0.248	0.230	0.235	0.220
128 k	0.229	0.251	0.217	0.242	0.225	0.230	0.214
256 k	0.228	0.249	0.216	0.240	0.222	0.228	0.212
512 k	0.239	0.260	0.226	0.251	0.231	0.236	0.224
1 M	0.243	0.262	0.230	0.254	0.235	0.241	0.227
2 M	0.240	0.261	0.227	0.251	0.232	0.237	0.222
4 M	0.240	0.261	0.227	0.252	0.230	0.232	0.221
8 M	0.245	0.266	0.232	0.256	0.233	0.240	0.228
16 M	0.255	0.275	0.241	0.264	0.240	0.245	0.233
32 M	0.277	0.302	0.266	0.288	0.258	0.267	0.253
64 M	0.312	0.356	0.309	0.331	0.294	0.306	0.293
128 M	0.324	0.375	0.324	0.347	0.312	0.321	0.311

Table A.10: Benchmark with *silesia8m* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.799	2.861	2.870	2.937	2.919	2.945	2.919
64 k	2.961	3.023	3.041	3.108	3.106	3.133	3.106
128 k	3.090	3.149	3.177	3.241	3.254	3.282	3.254
256 k	3.184	3.240	3.275	3.337	3.366	3.393	3.366
512 k	3.252	3.305	3.349	3.409	3.457	3.482	3.457
1 M	3.319	3.370	3.422	3.480	3.539	3.563	3.539
2 M	3.376	3.423	3.484	3.538	3.608	3.631	3.608
4 M	3.425	3.468	3.534	3.584	3.665	3.688	3.665
8 M	3.463	3.502	3.569	3.616	3.704	3.726	3.704
16 M	3.508	3.537	3.613	3.655	3.749	3.772	3.749
32 M	3.517	3.523	3.622	3.639	3.761	3.784	3.761
64 M	3.529	3.535	3.634	3.651	3.774	3.796	3.774
128 M	3.532	3.538	3.637	3.652	3.778	3.801	3.778
	Compression time (seconds)						
32 k	3.654	3.752	3.737	3.861	4.995	4.923	4.564
64 k	3.578	3.685	3.681	3.782	5.005	4.969	4.568
128 k	3.557	3.647	3.650	3.734	5.014	4.975	4.570
256 k	3.628	3.712	3.731	3.811	5.160	5.114	4.717
512 k	3.658	3.758	3.774	3.848	5.402	5.359	4.936
1 M	3.696	3.849	3.827	3.937	5.572	5.523	5.171
2 M	3.876	4.034	4.018	4.133	5.782	5.734	5.443
4 M	4.154	4.289	4.263	4.362	6.124	6.082	5.767
8 M	4.569	4.649	4.648	4.724	6.518	6.491	6.161
16 M	5.113	5.180	5.163	5.239	7.134	7.121	6.744
32 M	5.476	5.421	5.386	5.477	7.445	7.445	7.069
64 M	5.661	5.757	5.705	5.780	7.888	7.889	7.577
128 M	6.064	6.239	6.155	6.278	8.514	8.529	8.251
	Decompression time (seconds)						
32 k	0.237	0.251	0.224	0.245	0.227	0.227	0.216
64 k	0.228	0.245	0.215	0.237	0.220	0.222	0.211
128 k	0.219	0.237	0.206	0.230	0.213	0.217	0.203
256 k	0.218	0.235	0.207	0.229	0.213	0.217	0.204
512 k	0.224	0.242	0.214	0.234	0.217	0.222	0.211
1 M	0.222	0.240	0.211	0.234	0.217	0.221	0.208
2 M	0.220	0.236	0.209	0.227	0.213	0.217	0.204
4 M	0.220	0.238	0.208	0.228	0.211	0.216	0.204
8 M	0.222	0.240	0.212	0.230	0.216	0.220	0.211
16 M	0.226	0.240	0.216	0.233	0.221	0.224	0.215
32 M	0.238	0.250	0.224	0.247	0.230	0.240	0.221
64 M	0.256	0.270	0.241	0.259	0.243	0.251	0.238
128 M	0.259	0.278	0.257	0.267	0.253	0.256	0.244

Table A.11: Benchmark with *silesia8t* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	2.219	2.280	2.262	2.319	2.304	2.309	2.304
64 k	2.339	2.396	2.388	2.442	2.444	2.449	2.444
128 k	2.442	2.493	2.496	2.543	2.564	2.569	2.564
256 k	2.528	2.573	2.585	2.625	2.668	2.673	2.668
512 k	2.586	2.624	2.647	2.682	2.745	2.750	2.745
1 M	2.644	2.679	2.709	2.743	2.819	2.824	2.819
2 M	2.703	2.736	2.771	2.804	2.889	2.894	2.889
4 M	2.742	2.770	2.810	2.839	2.934	2.939	2.934
8 M	2.790	2.814	2.856	2.880	2.982	2.987	2.982
16 M	2.813	2.826	2.878	2.892	3.008	3.013	3.008
32 M	2.828	2.833	2.893	2.899	3.026	3.031	3.026
64 M	2.831	2.836	2.896	2.902	3.030	3.036	3.030
128 M	2.847	2.848	2.912	2.915	3.049	3.054	3.049
	Compression time (seconds)						
32 k	4.224	4.392	4.390	4.545	5.809	5.712	5.239
64 k	4.020	4.190	4.194	4.350	5.679	5.602	5.123
128 k	3.949	4.109	4.114	4.250	5.638	5.573	5.111
256 k	3.991	4.151	4.169	4.303	5.766	5.699	5.240
512 k	4.011	4.190	4.192	4.320	6.028	5.967	5.476
1 M	4.079	4.253	4.263	4.393	6.192	6.128	5.736
2 M	4.253	4.426	4.433	4.553	6.382	6.321	5.990
4 M	4.574	4.716	4.730	4.852	6.776	6.735	6.387
8 M	5.020	5.166	5.174	5.264	7.302	7.271	6.845
16 M	5.537	5.714	5.692	5.791	8.054	8.036	7.591
32 M	5.981	6.143	6.123	6.217	8.696	8.701	8.272
64 M	6.369	6.557	6.522	6.621	9.262	9.304	8.873
128 M	6.827	7.023	6.979	7.102	10.237	10.285	9.861
	Decompression time (seconds)						
32 k	0.288	0.310	0.280	0.306	0.289	0.288	0.274
64 k	0.278	0.301	0.270	0.299	0.278	0.282	0.264
128 k	0.267	0.290	0.259	0.287	0.266	0.272	0.256
256 k	0.268	0.291	0.260	0.286	0.265	0.272	0.252
512 k	0.275	0.299	0.265	0.295	0.271	0.279	0.261
1 M	0.274	0.299	0.266	0.296	0.271	0.279	0.261
2 M	0.267	0.293	0.258	0.288	0.262	0.269	0.253
4 M	0.268	0.292	0.260	0.283	0.260	0.266	0.248
8 M	0.273	0.297	0.264	0.290	0.263	0.271	0.255
16 M	0.299	0.320	0.288	0.315	0.285	0.290	0.275
32 M	0.330	0.350	0.314	0.336	0.310	0.321	0.302
64 M	0.346	0.365	0.332	0.352	0.320	0.330	0.314
128 M	0.385	0.427	0.373	0.402	0.357	0.367	0.356

Table A.12: Benchmark with *sources8* payload The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	3.311	3.343	3.397	3.429	3.458	3.476	3.458
64 k	3.539	3.565	3.639	3.667	3.721	3.741	3.721
128 k	3.746	3.763	3.857	3.880	3.958	3.980	3.958
256 k	3.901	3.911	4.024	4.043	4.146	4.169	4.146
512 k	4.009	4.015	4.145	4.165	4.290	4.315	4.290
1 M	4.100	4.104	4.249	4.268	4.413	4.438	4.413
2 M	4.170	4.173	4.330	4.348	4.512	4.537	4.512
4 M	4.213	4.216	4.379	4.397	4.581	4.607	4.581
8 M	4.277	4.280	4.447	4.465	4.666	4.692	4.666
16 M	4.335	4.338	4.511	4.529	4.750	4.777	4.750
32 M	4.369	4.371	4.552	4.568	4.815	4.841	4.815
64 M	4.421	4.423	4.610	4.626	4.885	4.911	4.885
128 M	4.493	4.495	4.688	4.705	4.981	5.008	4.981
	Compression time (seconds)						
32 k	3.878	4.060	4.062	4.185	5.479	5.424	4.922
64 k	3.693	3.880	3.885	3.998	5.435	5.385	4.857
128 k	3.608	3.786	3.791	3.898	5.394	5.349	4.815
256 k	3.655	3.819	3.834	3.935	5.539	5.497	4.944
512 k	3.661	3.822	3.843	3.945	5.779	5.727	5.122
1 M	3.720	3.886	3.904	3.991	5.949	5.902	5.350
2 M	3.864	4.035	4.038	4.142	6.126	6.082	5.601
4 M	4.091	4.235	4.256	4.333	6.437	6.400	5.888
8 M	4.455	4.589	4.597	4.671	6.832	6.805	6.251
16 M	5.034	5.181	5.143	5.237	7.654	7.661	7.049
32 M	5.587	5.725	5.745	5.821	8.737	8.760	8.161
64 M	5.943	6.095	6.089	6.166	9.501	9.576	8.942
128 M	6.339	6.482	6.489	6.556	10.417	10.505	9.813
	Decompression time (seconds)						
32 k	0.246	0.274	0.241	0.270	0.244	0.248	0.232
64 k	0.236	0.265	0.231	0.263	0.236	0.243	0.224
128 k	0.228	0.256	0.221	0.251	0.225	0.233	0.217
256 k	0.222	0.249	0.216	0.246	0.217	0.225	0.210
512 k	0.221	0.247	0.213	0.243	0.214	0.224	0.207
1 M	0.229	0.255	0.223	0.251	0.221	0.231	0.217
2 M	0.229	0.254	0.222	0.250	0.221	0.230	0.218
4 M	0.227	0.254	0.221	0.250	0.218	0.226	0.214
8 M	0.231	0.257	0.225	0.252	0.220	0.229	0.216
16 M	0.252	0.274	0.246	0.262	0.235	0.246	0.232
32 M	0.332	0.356	0.315	0.328	0.288	0.302	0.287
64 M	0.376	0.401	0.351	0.372	0.325	0.343	0.320
128 M	0.423	0.469	0.391	0.421	0.361	0.378	0.358

Table A.13: Benchmark with *zero8* payload. The best results for each block size are highlighted.

	Data compressor design						
	greedy	**greedy-opt**	**nongreedy**	**nongreedy-opt**	**mincost**	**mincost-opt**	**dp-mincost**
Block size	Compression ratio						
32 k	61	1130	61	1130	61	60	61
64 k	63	2260	63	2260	63	62	63
128 k	63	4519	63	4519	63	63	63
256 k	64	9024	64	9024	64	64	64
512 k	64	18044	64	18044	64	64	64
1 M	64	35881	64	35881	64	64	64
2 M	64	71685	64	71685	64	64	64
4 M	64	142857	64	142857	64	64	64
8 M	64	284091	64	284091	64	64	64
16 M	64	561798	64	561798	64	64	64
32 M	64	1098901	64	1098901	64	64	64
64 M	64	1612903	64	1612903	64	64	64
128 M	64	3030303	64	3030303	64	64	64
	Compression time (seconds)						
32 k	1.108	1.707	1.167	1.735	1.819	1.877	1.626
64 k	1.107	1.433	1.167	1.454	1.810	1.872	1.643
128 k	1.105	1.298	1.162	1.316	1.805	1.851	1.640
256 k	1.148	1.283	1.220	1.304	1.871	1.906	1.715
512 k	1.150	1.248	1.225	1.265	1.896	1.913	1.747
1 M	1.165	1.236	1.222	1.248	1.924	1.928	1.763
2 M	1.172	1.240	1.230	1.252	1.931	1.942	1.793
4 M	1.191	1.256	1.249	1.269	1.980	1.998	1.837
8 M	1.233	1.294	1.288	1.307	2.012	2.025	1.859
16 M	1.254	1.373	1.312	1.330	2.046	2.073	1.899
32 M	1.306	1.380	1.374	1.384	2.142	2.185	2.005
64 M	1.425	1.502	1.483	1.505	2.313	2.329	2.156
128 M	1.525	1.627	1.579	1.656	2.457	2.497	2.319
	Decompression time (seconds)						
32 k	0.061	0.060	0.061	0.060	0.061	0.061	0.060
64 k	0.059	0.057	0.059	0.057	0.058	0.059	0.058
128 k	0.060	0.057	0.058	0.057	0.058	0.058	0.058
256 k	0.058	0.056	0.058	0.056	0.057	0.057	0.057
512 k	0.057	0.056	0.057	0.056	0.057	0.057	0.057
1 M	0.058	0.056	0.058	0.056	0.057	0.057	0.059
2 M	0.059	0.058	0.059	0.058	0.059	0.059	0.059
4 M	0.061	0.060	0.061	0.060	0.061	0.061	0.061
8 M	0.065	0.063	0.064	0.063	0.065	0.064	0.064
16 M	0.068	0.067	0.068	0.066	0.068	0.068	0.068
32 M	0.073	0.071	0.073	0.071	0.073	0.075	0.073
64 M	0.081	0.081	0.082	0.082	0.081	0.083	0.084
128 M	0.092	0.091	0.090	0.093	0.092	0.090	0.090

Appendix B SALZ micro-benchmarks

Lcp-comparison

Table B.1: The effect of performing lcp-comparison using multi-byte words on Lempel-Ziv factorization time (seconds), when only the positions used in final encoding are factorized. Benchmarked on *greedy* compressor design and *silesia* payload. The best results for each block size are highlighted.

		8B words		16B words		32B words	
Block size	**Base**	**Time**	**Change**	**Time**	**Change**	**Time**	**Change**
32 k	1.117	0.686	-38.61%	0.709	-36.53%	0.722	-35.35%
64 k	1.080	0.671	-37.86%	0.696	-35.58%	0.732	-32.20%
128 k	1.061	0.659	-37.89%	0.677	-36.17%	0.707	-33.37%
256 k	1.037	0.652	-37.14%	0.668	-35.59%	0.714	-31.14%
512 k	1.064	0.698	-34.37%	0.711	-33.24%	0.754	-29.11%
1 M	1.090	0.742	-31.94%	0.755	-30.75%	0.790	-27.47%
2 M	1.123	0.795	-29.17%	0.812	-27.69%	0.848	-24.47%
4 M	1.154	0.834	-27.72%	0.850	-26.33%	0.890	-22.82%
8 M	1.201	0.889	-25.94%	0.907	-24.47%	0.951	-20.81%
16 M	1.337	1.057	-20.90%	1.076	-19.51%	1.143	-14.50%
32 M	1.666	1.418	-14.88%	1.488	-10.70%	1.600	-4.01%
64 M	1.924	1.694	-11.97%	1.869	-2.85%	1.901	-1.18%
128 M	2.149	1.987	-7.51%	2.161	0.58%	2.287	6.46%

Table B.2: The effect of performing lcp-comparison using multi-byte words on Lempel-Ziv factorization time (seconds), when all text positions are factorized. Benchmarked on *dp-mincost* compressor design and *silesia* payload. The best results for each block size are highlighted.

		8B words		16B words		32B words	
Block size	**Base**	**Time**	**Change**	**Time**	**Change**	**Time**	**Change**
32 k	8.463	1.946	-77.00%	1.760	-79.20%	1.635	-80.68%
64 k	9.518	2.127	-77.66%	1.875	-80.30%	1.740	-81.71%
128 k	10.800	2.348	-78.26%	2.027	-81.23%	1.817	-83.18%
256 k	12.187	2.513	-79.38%	2.164	-82.24%	1.894	-84.46%
512 k	12.923	2.785	-78.45%	2.377	-81.61%	2.072	-83.96%
1 M	13.972	3.074	-78.00%	2.603	-81.37%	2.242	-83.96%
2 M	14.614	3.305	-77.39%	2.776	-81.01%	2.383	-83.70%
4 M	15.431	3.485	-77.42%	2.949	-80.89%	2.545	-83.51%
8 M	17.006	3.772	-77.82%	3.176	-81.32%	2.726	-83.97%
16 M	20.251	4.222	-79.15%	3.626	-82.09%	3.236	-84.02%
32 M	20.228	4.928	-75.64%	4.176	-79.36%	3.892	-80.76%
64 M	20.881	5.327	-74.49%	4.706	-77.46%	4.374	-79.05%
128 M	21.189	5.628	-73.44%	4.910	-76.83%	4.636	-78.12%

Table B.3: The effect on reducing the number of symbol comparisons on Lempel-Ziv factorization time (seconds), when all text positions are factorized. Benchmarked on *dp-mincost* compressor design and *silesia* payload.

Block size	Base	Optimized	Change
32 k	8.463	2.544	-69.93%
64 k	9.518	2.560	-73.11%
128 k	10.800	2.594	-75.99%
256 k	12.187	2.584	-78.80%
512 k	12.923	2.660	-79.41%
1 M	13.972	2.772	-80.16%
2 M	14.614	2.846	-80.53%
4 M	15.431	2.912	-81.13%
8 M	17.006	3.005	-82.33%
16 M	20.251	3.265	-83.88%
32 M	20.228	3.701	-81.70%
64 M	20.881	4.094	-80.40%
128 M	21.189	4.294	-79.73%

Table B.4: The effect of performing lcp-comparison using multi-byte words on Lempel-Ziv factorization time (seconds), when all text positions are factorized with reduced number of symbol comparisons. Benchmarked on *dp-mincost* compressor design and *silesia* payload. The best results for each block size are highlighted.

		8B words		16B words		32B words	
Block size	Base	Time	Change	Time	Change	Time	Change
32 k	2.544	1.462	-42.52%	1.730	-32.03%	1.886	-25.87%
64 k	2.560	1.490	-41.79%	1.767	-30.95%	1.955	-23.61%
128 k	2.594	1.557	-39.95%	1.829	-29.49%	1.978	-23.75%
256 k	2.584	1.589	-38.51%	1.895	-26.66%	2.042	-20.97%
512 k	2.660	1.777	-33.20%	2.047	-23.06%	2.207	-17.05%
1 M	2.772	1.948	-29.71%	2.213	-20.15%	2.408	-13.14%
2 M	2.846	2.073	-27.17%	2.345	-17.58%	2.548	-10.45%
4 M	2.912	2.173	-25.36%	2.451	-15.84%	2.673	-8.20%
8 M	3.005	2.348	-21.88%	2.609	-13.18%	2.856	-4.97%
16 M	3.265	2.829	-13.36%	3.148	-3.57%	3.449	5.64%
32 M	3.701	3.774	1.97%	4.133	11.67%	4.528	22.35%
64 M	4.094	4.395	7.37%	4.885	19.34%	5.535	35.22%
128 M	4.294	4.993	16.28%	5.473	27.45%	5.957	38.72%

Factor offset reuse

Table B.5: Cache hit statistics measured for factor offset reuse. The statistics contain the total cache hit rate and the distribution of hits per cache position. The statistics were obtained using 8-slot Move-to-Front cache on *dp-mincost* design and *silesia* payload.

		Per position							
Block size	**Total**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
32 k	0.096	0.406	0.217	0.112	0.091	0.060	0.046	0.036	0.032
64 k	0.086	0.405	0.222	0.110	0.092	0.060	0.045	0.035	0.031
128 k	0.078	0.403	0.227	0.109	0.089	0.064	0.044	0.034	0.030
256 k	0.073	0.400	0.232	0.108	0.088	0.067	0.043	0.033	0.029
512 k	0.070	0.397	0.235	0.109	0.089	0.066	0.042	0.033	0.028
1 M	0.067	0.395	0.237	0.110	0.090	0.066	0.042	0.032	0.028
2 M	0.064	0.393	0.237	0.111	0.091	0.066	0.041	0.032	0.028
4 M	0.063	0.389	0.238	0.112	0.092	0.066	0.042	0.032	0.028
8 M	0.061	0.383	0.239	0.114	0.093	0.067	0.042	0.032	0.028
16 M	0.060	0.381	0.239	0.115	0.095	0.068	0.042	0.032	0.028
32 M	0.059	0.380	0.239	0.115	0.095	0.068	0.042	0.032	0.028
64 M	0.058	0.378	0.240	0.116	0.096	0.068	0.042	0.032	0.029
128 M	0.058	0.378	0.240	0.116	0.096	0.068	0.042	0.032	0.029

Table B.6: The effect of factor offset reuse optimization on compression ratio and encoding and decoding time (seconds). Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

	Before	**After**	**Change**
Block size	Compression ratio		
32 k	2.561	2.581	0.78%
64 k	2.674	2.693	0.71%
128 k	2.768	2.787	0.69%
256 k	2.847	2.866	0.67%
512 k	2.904	2.923	0.65%
1 M	2.961	2.979	0.61%
2 M	3.015	3.032	0.56%
4 M	3.051	3.067	0.52%
8 M	3.083	3.098	0.49%
16 M	3.111	3.126	0.48%
32 M	3.122	3.136	0.45%
64 M	3.131	3.144	0.42%
128 M	3.119	3.133	0.45%
	Encoding time		
32 k	0.397	0.443	11.55%
64 k	0.393	0.427	8.66%
128 k	0.365	0.408	11.69%
256 k	0.351	0.395	12.57%
512 k	0.348	0.384	10.35%
1 M	0.334	0.367	9.81%
2 M	0.322	0.361	12.18%
4 M	0.318	0.369	16.09%
8 M	0.327	0.368	12.54%
16 M	0.354	0.378	6.89%
32 M	0.348	0.380	9.35%
64 M	0.332	0.367	10.61%
128 M	0.334	0.362	8.28%
	Decoding time		
32 k	0.423	0.438	3.61%
64 k	0.417	0.437	4.84%
128 k	0.407	0.425	4.42%
256 k	0.393	0.419	6.52%
512 k	0.387	0.413	6.72%
1 M	0.376	0.404	7.65%
2 M	0.389	0.396	1.87%
4 M	0.373	0.397	6.57%
8 M	0.381	0.404	5.99%
16 M	0.414	0.437	5.57%
32 M	0.476	0.493	3.62%
64 M	0.542	0.572	5.63%
128 M	0.593	0.611	3.14%

Choosing optimal integer codes

Table B.7: The overhead of Golomb-Rice optimization stage on compression time. Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

Block size	**Total** (ms)	**Per segment** (μs)
32 k	12.127	1.875
64 k	9.658	2.986
128 k	7.530	4.657
256 k	5.402	6.677
512 k	3.860	9.531
1 M	2.465	12.143
2 M	1.598	15.667
4 M	1.122	22.000
8 M	0.780	30.000
16 M	0.515	39.615
32 M	0.337	48.143
64 M	0.170	42.500
128 M	0.104	52.000

Table B.8: The effect of Golomb-Rice optimization on compression ratio. Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

Block size	**Base**	**Optimized**	**Change**
32 k	2.508	2.561	2.11%
64 k	2.625	2.674	1.87%
128 k	2.725	2.768	1.58%
256 k	2.810	2.847	1.32%
512 k	2.871	2.904	1.15%
1 M	2.930	2.961	1.06%
2 M	2.985	3.015	1.01%
4 M	3.026	3.051	0.83%
8 M	3.065	3.083	0.59%
16 M	3.096	3.111	0.48%
32 M	3.107	3.122	0.48%
64 M	3.119	3.131	0.38%
128 M	3.119	3.119	0.00%

Memory reuse

Table B.9: The time (milliseconds) taken by memory resource allocation for compressing a single segment. Benchmarked with *dp-mincost* compressor design.

Block size	Allocation time
32 k	0.101
64 k	0.138
128 k	0.244
256 k	0.461
512 k	0.817
1 M	1.329
2 M	2.517
4 M	4.518
8 M	7.965
16 M	16.363
32 M	34.869
64 M	68.899
128 M	132.666

Memory access patterns

Table B.10: The effect of optimizing memory access patterns on compressions time (seconds), excluding suffix array construction time. Benchmarked on *dp-mincost* compressor design and *silesia* payload.

Block size	Base	Optimized	Change
32 k	4.644	4.605	-0.82%
64 k	4.765	4.870	2.21%
128 k	4.833	5.194	7.45%
256 k	5.318	5.573	4.80%
512 k	5.607	5.977	6.61%
1 M	5.979	6.251	4.54%
2 M	6.875	6.603	-3.96%
4 M	7.666	7.217	-5.86%
8 M	8.434	7.212	-14.49%
16 M	9.325	7.954	-14.70%
32 M	10.349	9.020	-12.84%
64 M	11.492	9.954	-13.39%
128 M	12.737	10.934	-14.15%

Integer code optimizations

Table B.11: The effect of integer code performance optimizations on encoding and decoding time (seconds). Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

	Single bits	**Multiple bits**	**Change**
Block size	Encoding time		
32 k	0.516	0.445	-13.73%
64 k	0.503	0.413	-17.85%
128 k	0.502	0.422	-15.98%
256 k	0.535	0.381	-28.67%
512 k	0.578	0.447	-22.73%
1 M	0.665	0.542	-18.46%
2 M	0.682	0.529	-22.45%
4 M	0.713	0.533	-25.26%
8 M	0.705	0.539	-23.54%
16 M	0.700	0.546	-22.03%
32 M	0.703	0.556	-20.91%
64 M	0.734	0.587	-20.07%
128 M	0.746	0.602	-19.31%
	Decoding time		
32 k	0.425	0.450	5.79%
64 k	0.428	0.428	0.00%
128 k	0.431	0.436	1.14%
256 k	0.427	0.432	1.18%
512 k	0.422	0.401	-5.01%
1 M	0.420	0.388	-7.46%
2 M	0.417	0.407	-2.37%
4 M	0.428	0.432	0.82%
8 M	0.445	0.386	-13.41%
16 M	0.471	0.422	-10.49%
32 M	0.505	0.448	-11.13%
64 M	0.573	0.517	-9.74%
128 M	0.607	0.553	-8.99%

Reduced branching

Table B.12: The effect of reducing branching on the time (seconds) spent in backtracking stage of *mincost* compressor design. Benchmarked on *silesia* payload.

Block size	Base	Optimized	Change
32 k	0.239	0.191	-20.19%
64 k	0.232	0.185	-20.11%
128 k	0.227	0.175	-22.73%
256 k	0.230	0.199	-13.42%
512 k	0.379	0.345	-8.88%
1 M	0.470	0.447	-4.98%
2 M	0.425	0.405	-4.72%
4 M	0.400	0.374	-6.60%
8 M	0.402	0.370	-7.87%
16 M	0.419	0.392	-6.38%
32 M	0.445	0.415	-6.80%
64 M	0.455	0.424	-6.87%
128 M	0.471	0.447	-4.93%

Compile time branch prediction

Table B.13: The effect of using compile time branch prediction in factor offset reuse optimization on total encoding and decoding time (seconds). Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

	Before	**After**	**Change**
Block size	Encoding time		
32 k	0.443	0.458	3.40%
64 k	0.427	0.441	3.26%
128 k	0.408	0.415	1.78%
256 k	0.395	0.401	1.41%
512 k	0.384	0.393	2.37%
1 M	0.367	0.372	1.51%
2 M	0.361	0.384	6.29%
4 M	0.369	0.380	2.83%
8 M	0.368	0.365	-0.58%
16 M	0.378	0.391	3.44%
32 M	0.380	0.384	1.08%
64 M	0.367	0.371	1.13%
128 M	0.362	0.377	4.12%
	Decoding time		
32 k	0.438	0.428	-2.37%
64 k	0.437	0.425	-2.77%
128 k	0.425	0.437	2.82%
256 k	0.419	0.408	-2.69%
512 k	0.413	0.406	-1.82%
1 M	0.404	0.404	-0.17%
2 M	0.396	0.391	-1.40%
4 M	0.397	0.397	-0.04%
8 M	0.404	0.401	-0.63%
16 M	0.437	0.435	-0.40%
32 M	0.493	0.495	0.43%
64 M	0.572	0.605	5.69%
128 M	0.611	0.627	2.51%

Table B.14: The effect of using compile time branch prediction on time (seconds) spent in Golomb-Rice optimization stage. Benchmarked with *nongreedy-opt* compressor design and *silesia* payload.

Block size	Before	After	Change
32 k	12.127	12.176	0.40%
64 k	9.658	9.548	-1.14%
128 k	7.530	7.101	-5.70%
256 k	5.402	5.383	-0.35%
512 k	3.860	3.759	-2.62%
1 M	2.465	2.394	-2.88%
2 M	1.598	1.603	0.31%
4 M	1.122	1.080	-3.74%
8 M	0.780	0.759	-2.69%
16 M	0.515	0.528	2.52%
32 M	0.337	0.329	-2.37%
64 M	0.170	0.175	2.94%
128 M	0.104	0.095	-8.65%

Loop unrolling

Table B.15: The effect of loop unrolling on combined time (seconds) spent in minimum-cost optimization and encoding stages of *mincost* compressor design. Benchmarked on *silesia* payload.

Block size	Base	Optimized	Change
32 k	2.061	2.099	1.89%
64 k	2.176	2.238	2.86%
128 k	2.208	2.156	-2.37%
256 k	2.327	2.281	-1.97%
512 k	2.521	2.489	-1.29%
1 M	2.668	2.671	0.10%
2 M	2.724	2.749	0.92%
4 M	2.957	2.811	-4.92%
8 M	2.911	2.867	-1.50%
16 M	3.055	2.951	-3.41%
32 M	3.132	2.929	-6.47%
64 M	3.188	3.029	-5.00%
128 M	3.209	3.055	-4.80%

Copying in words

Table B.16: The effect of copying in words on decoding time (seconds). Benchmarked on *dp-mincost* compressor design and *silesia* payload.

Block size	Bytes	Words	Change
32 k	0.561	0.443	-21.07%
64 k	0.559	0.440	-21.35%
128 k	0.567	0.424	-25.22%
256 k	0.537	0.431	-19.77%
512 k	0.537	0.421	-21.53%
1 M	0.527	0.417	-20.89%
2 M	0.521	0.420	-19.43%
4 M	0.526	0.417	-20.75%
8 M	0.539	0.452	-16.05%
16 M	0.578	0.482	-16.60%
32 M	0.677	0.568	-16.10%
64 M	0.808	0.680	-15.87%
128 M	0.899	0.779	-13.25%

Appendix C Other benchmarks

Table C.1: Benchmarks on third-party data compressors include the de facto standard data compressors for Unix-like operating systems (`gzip`, `bzip2` and `xz`) and two well-known modern alternatives (`lz4` and `zstd`). The data compressors were benchmarked with their default, minimum and maximum compression levels. `zstd` offers additional compression levels outside of the default range, which were tested in addition. The benchmarks were performed on *silesia* payload.

	Compression ratio	**Compression time (s)**	**Decompression time (s)**
Compression level		`gzip`	
(min) `-1`	2.739	3.196	1.394
(default) `-6`	3.106	7.644	1.286
(max) `-9`	3.133	16.926	1.277
		`bzip2`	
(min) `-1`	3.504	16.176	5.752
(max, default) `-9`	3.883	18.206	6.016
		`xz`	
(min) `-1`	3.628	12.242	3.592
(*default*) `-6`	4.308	91.320	3.015
(max) `-9`	4.347	106.080	3.006
		`lz4`	
(min, default) `-1`	2.100	0.635	0.369
(max) `-12`	2.739	22.268	0.351
		`zstd`	
`--fast`	2.433	1.154	0.302
(min) `-1`	2.881	1.338	0.367
(default) `-3`	3.177	1.635	0.405
(max) `-19`	3.979	103.199	0.428
`--ultra -22`	4.019	121.869	0.449