HP-SBWT
High Performance SBWT Compression

Fulvio Serao, Marco Fusco, Antonio Garofalo, Marco Palmisciano
Università degli Studi di Salerno, *Dipartimento di Informatica*

Abstract

Efficient text compression is a critical area in data storage and transmission, aiming to reduce file sizes while preserving data integrity. This paper introduces a novel approach to text compression that combines a scrambled Burrows-Wheeler Transform (SBWT) with three compression techniques: Huffman coding, Bzip2, LZW, and Arithmetic Coding. The SBWT is enhanced through a custom character ordering mechanism driven by a user-defined key, ensuring data scrambling for additional security. Following the transformation, a Move-to-Front (MTF) encoding step reduces data entropy, enabling effective compression with either Huffman, Lempel-Ziv-Welch (LZW), or Arithmetic Coding.
The inclusion of Arithmetic Coding provides an additional mechanism to achieve higher compression ratios for datasets with uneven symbol distributions, improving over Huffman in these scenarios. Its precision-based encoding further complements the modular design of the proposed system. However, its computational overhead requires careful optimization, making it suitable for high-efficiency applications where maximum compression is a priority.
The implementation leverages multi-processing to handle large files, dividing input data into blocks for parallel compression and decompression. Experimental results demonstrate the efficiency of the proposed system across various text file types, showcasing significant size reduction, competitive processing times, and flexibility in algorithm selection. This approach not only achieves robust compression but also introduces an additional layer of data obfuscation, making it suitable for applications where both storage efficiency and data privacy are priorities.

1 Introduction

Data compression is an essential component of modern computing, enabling efficient storage and transmission of large datasets. Over the years, various compression algorithms have been developed, each tailored to address specific challenges and data types. Traditional methods like the Burrows-Wheeler Transform (BWT) [1], Huffman coding [2], and Arithmetic Coding [3] have proven to be effective for reducing file sizes while maintaining data integrity. Huffman coding assigns variable-length codes to symbols based on their frequency, offering a balance between compression efficiency and computational simplicity.
Building on these foundational techniques, algorithms like Bzip2 [4] and LZW [5] have emerged to address additional compression needs. Bzip2 combines the Burrows-Wheeler Transform with Move-to-Front encoding [6] and entropy coding to achieve high compression ratios for text-heavy or repetitive data. Following Move-to-Front encoding, Run Length Encoding (RLE) [7, 8] can be applied to further enhance compression, particularly effective for sequences with extended runs of identical characters.This methodology excels at handling datasets with significant redundancy, making it a popular choice for archival purposes. Meanwhile, LZW, a dictionary-based algorithm, dynamically builds a dictionary of recurring patterns, allowing for rapid compression and decompression. Its speed and simplicity make it well-suited for real-time applications, such as GIF compression and lightweight data storage.
Despite the strengths of these methods, challenges remain in balancing compression efficiency, speed, and adaptability across diverse data types and use cases. For example, entropy-based methods like Huffman coding excel at compressing data with predictable symbol distributions but can falter when faced with highly non-uniform datasets. Arithmetic Coding, by contrast, provides a more flexible approach to entropy coding, as it encodes data using probabilistic models to assign fractional bits to symbols. This results in superior compression ratios, especially in scenarios where symbol frequencies vary significantly.
The precision of Arithmetic Coding comes at a computational cost, as its operations require managing ranges and probabilities with higher accuracy than traditional methods like Huffman coding. Recent advances, however, including modular designs and parallel processing, enable the integration of Arithmetic Coding into scalable systems without significant performance degradation. This paper introduces a modular compression pipeline that leverages the strengths of SBWT, MTF encoding, and multiple compression techniques, including Arithmetic Coding, to optimize both efficiency and flexibility.
The remainder of this paper is structured as follows. Section 2 provides background information on foundational compression techniques, including the Burrows-Wheeler Transform (BWT), Huffman coding, Bzip2, LZW, and Arithmetic Coding.

Section 3 outlines the methodology used in the proposed system, focusing on the scrambled Burrows-Wheeler Transform (SBWT) and its integration with entropy coding. Section 4 describes the experimental setup employed to evaluate the system's performance. Section 5 presents the results of the evaluation, analyzing the system's performance in terms of compression ratio, execution time, and scalability. Section 6 discusses key trade-offs observed during the study and explores potential future improvements. Finally, Section 7 summarizes the contributions of this work and suggests directions for further research.

2 Background

The development and refinement of text compression technologies have significantly impacted data storage and transmission across various applications. This section delves into the foundational techniques that form the bedrock of our proposed compression system. Each technology discussed here has been integral to the evolution of data compression strategies and serves a specific role in enhancing the system's efficiency, security, and adaptability.

2.1 Key Handling with Counter Mode

Key management is a critical component of the system, especially in ensuring data security during the scrambled Burrows-Wheeler Transform (SBWT). The proposed system leverages the Counter Mode (CTR) of operation for key derivation to guarantee confidentiality and **Chosen Plaintext Attack (CPA)** security [9].

In the Counter Mode, the user-provided master key is combined with a unique nonce (e.g., block number) to generate a sequence of sub-keys. Each sub-key is derived by encrypting the concatenated master key and nonce using a cryptographic hash function like SHA-256 [10]. This ensures that every block of data is scrambled with a distinct, deterministic key.

The use of CTR mode offers several advantages:

- **Deterministic Key Derivation**: Each block is processed independently, with its unique sub-key, allowing for parallelism in the compression and decompression processes.
- **CPA Security**: By ensuring that every sub-key is unique and non-repeating, the system is resilient to cryptanalytic attacks that exploit key reuse.
- **Efficiency and Scalability**: The lightweight nature of CTR mode ensures minimal overhead while maintaining robust security properties.

The derived sub-keys are encoded in Base64 format for safe storage and transmission. This mechanism not only enhances the system's security but also aligns with the modular design, allowing seamless integration of cryptographic processes into the compression pipeline.

2.2 Other Techniques

The other foundational techniques in the system include:

- **Scrambled Burrows-Wheeler Transform (SBWT)**: The SBWT is an enhancement of the classic Burrows-Wheeler Transform (BWT), designed to introduce a layer of data obfuscation. By using a key-driven character ordering mechanism, the SBWT scrambles the input data before compression and provides privacy [11]. This process enhances data security by making it difficult to reconstruct the original dataset without the correct key. The SBWT also prepares the data for subsequent entropy reduction steps, such as Move-to-Front encoding, by clustering similar characters together.

- **Move-to-Front (MTF) Encoding**: Move-to-Front encoding is a preprocessing step that reduces data entropy by reordering symbols based on their frequency of use. It maintains a dynamically updated list of symbols, placing recently used characters at the front [6]. This transformation increases the efficiency of entropy-based compression methods by creating sequences of repeated symbols, which are more compressible.

- **Run Length Encoding (RLE):** RLE is a data compression method that condenses sequences of identical data elements into a single data value and count. This technique is particularly effective for data with long sequences of repeated characters or symbols. By recording only the length of sequences and the repeated character, RLE can significantly reduce the size of text or data files with high redundancy levels. Its simplicity offers fast compression and decompression, making it suitable for applications where processing speed is crucial. However, its efficiency decreases with less repetitive or more complex data.

- **Huffman Coding**: Huffman coding is a widely used entropy-based compression technique that assigns variable-length binary codes to symbols based on their frequencies [2]. Symbols that occur more frequently are given shorter codes, while less frequent symbols receive longer ones. This approach strikes a balance between compression efficiency and computational simplicity, making it suitable for a wide range of applications. Huffman coding works best with data that has a relatively uniform distribution of symbol probabilities.

- **Bzip2**: Bzip2 is a compression method that integrates the Burrows-Wheeler Transform (BWT), Move-to-Front encoding, and Huffman coding. It excels at compressing text-heavy or repetitive datasets by rearranging characters into runs of similar symbols through the BWT. The subsequent MTF encoding further reduces entropy, allowing Huffman coding to achieve superior compression ratios. Bzip2 is particularly effective for archival storage where high compression efficiency is prioritized over computational speed [4].

- **LZW (Lempel-Ziv-Welch)**: LZW is a dictionary-based compression algorithm that dynamically builds a dictionary of sequences encountered in the input data [5]. Each sequence is assigned a fixed-length code, allowing efficient compression of repetitive patterns. LZW is computationally lightweight and fast, making it ideal for real-time applications and scenarios where low latency is crucial, such as image compression in formats like GIF.
- **Arithmetic Coding**: Arithmetic Coding is an advanced entropy-based technique that encodes an entire message as a single fractional value between 0 and 1 [3, 12]. Unlike Huffman coding, which uses discrete binary codes, Arithmetic Coding assigns ranges to symbols based on their probabilities. This process narrows the range recursively as symbols are processed, with the final range representing the encoded message. Arithmetic Coding is especially effective for datasets with skewed symbol distributions, as it provides finer-grained compression by leveraging precise probability modeling. However, its computational requirements for managing ranges and probabilities are higher, making it most effective in scenarios where maximum compression efficiency is critical.

3 Proposed Methodology

The implementation of our proposed text compression system is designed around a pipeline that integrates multiple techniques to achieve optimal efficiency, security, and scalability. The following are the key components of the implementation process, detailed below.

3.1 Entropy Reduction with MTF Encoding

The compression pipeline begins with the scrambled Burrows-Wheeler Transform (SBWT), which reorganizes the input data into clusters of similar characters. To further reduce entropy, Move-to-Front (MTF) encoding is applied [6]. MTF encoding reorders symbols based on their frequency of usage, placing the most recently used characters at the front of the list. This transformation prepares the data for subsequent entropy-based compression techniques by creating long runs of repeated symbols, which are more efficiently compressed.

3.2 Compression Through Entropy Coding

After reducing entropy with SBWT and MTF, the system applies one of several entropy coding techniques to compress the data. These include:

- **Huffman Coding**: This method assigns variable-length codes to symbols based on their frequencies [2]. It is well-suited for datasets with uniform or predictable symbol distributions, offering a balance between speed and compression efficiency.
- **Bzip2**: This compression algorithm integrates the benefits of BWT, MTF, and Huffman coding. It is especially effective for repetitive or structured data, achieving high compression ratios at the cost of additional computational overhead [4].
- **LZW**: LZW uses a dictionary-based approach to encode sequences of symbols. It dynamically builds a dictionary of patterns encountered in the data, making it lightweight and fast for real-time applications.
- **Arithmetic Coding**: Arithmetic Coding represents the input message as a single fractional value between 0 and 1, based on the probabilities of each symbol [3, 12]. Unlike Huffman coding, which uses discrete codes, Arithmetic Coding achieves finer compression granularity by narrowing the range recursively as symbols are processed. This makes it particularly effective for datasets with highly skewed symbol distributions. The implementation leverages the modular design of the pipeline to seamlessly integrate Arithmetic Coding where maximum compression efficiency is required. However, due to its higher computational demands, it is selectively applied based on data characteristics.

3.3 Scalable Processing in Parallel

To handle large datasets efficiently, the system processes data in fixed-size blocks (64 KB by default). Each block is compressed independently, allowing for parallel processing using Python's `concurrent.futures.ProcessPoolExecutor`. The parallel design ensures scalability by distributing the computational load across multiple CPU cores.
During compression, each block undergoes the SBWT, MTF encoding, and the selected entropy coding technique independently. This modularity enables flexibility in choosing the best algorithm for specific data characteristics. For decompression, the process is reversed, with blocks reconstructed and combined in the correct order to restore the original file.

3.4 Secure Key Management

The SBWT relies on a user-provided key to derive a unique scrambling pattern for each block. The system uses a cryptographic hash function (e.g., SHA-256) in conjunction with Counter Mode (CTR) to generate block-specific sub-keys [9,

10]. Each sub-key is deterministic yet unique, ensuring data confidentiality and resistance to **chosen plaintext attacks (CPA)**.
Sub-keys are encoded in Base64 format for safe storage and transmission. This mechanism enhances both the security and scalability of the system, allowing parallel processing without compromising key integrity.

3.5 Data Serialization

The system employs efficient serialization techniques to store intermediate results during compression. Metadata, such as SBWT pointers, MTF symbol tables and RLE output, is saved alongside compressed data to facilitate seamless decompression. The use of modular serialization ensures that each block can be processed independently, further supporting the parallel design.

3.6 Integration of Arithmetic Coding

The modular pipeline allows the selective integration of Arithmetic Coding based on the input data's characteristics. For datasets with highly non-uniform symbol distributions, Arithmetic Coding provides significant improvements in compression ratio over Huffman coding. The implementation handles the computational overhead of managing precise ranges through optimized algorithms and parallel processing.
During compression, Arithmetic Coding processes the data block by block, updating probability distributions dynamically. At the end of each block, the encoded fractional range is serialized into a compact format. For decompression, the process reconstructs the original data by reversing the encoding operations and leveraging the stored probabilities.

4 Implementation Details

The proposed compression system is implemented in Python, leveraging modularity and parallelism to achieve scalability and efficiency. This section describes the key components and their functionalities, outlining their integration within the compression pipeline.

4.1 Core Modules

The system is divided into several modules, each responsible for a specific stage of the compression and decompression processes:

- **sbwt.py**: Implements the Scrambled Burrows-Wheeler Transform (SBWT). This module includes custom character order generation using a key, suffix array construction, and the SBWT transformation. The decoding process reverses these steps to reconstruct the original data.
- **mtf.py**: Handles Move-to-Front (MTF) encoding and decoding. It provides utilities to manage the dynamically updated symbol lists and to save or load symbol tables for consistency during decompression [6].
- **rle.py**: Handles Run Length Encoding (RLE) compression and decompression. This module is designed to compress sequences of repeated characters efficiently. It includes functions to encode and decode lists of integers, optimizing storage for data with high redundancy levels. The '$flush\_run$' function manages the output of encoded data, ensuring that sequences are represented compactly, especially handling special cases for frequently occurring characters. The encoding process compacts consecutive identical characters into a count and a symbol, while the decoding process reconstructs the original sequence from this compact form.
- **benchmark.py**: Manages the benchmarking of various compression algorithms including bzip2, Huffman, LZW, and Arithmetic Coding. This module orchestrates a series of compression and decompression tests, evaluating performance based on speed, efficiency, and reliability. The benchmarking process is logged in detail, including statistics on the number of tests performed, success and failure rates, and efficiency of data compression under different modes. It also handles error checking and validation of encryption keys, ensuring robust and reliable benchmarking sessions. The statistical data is captured in a structured format, allowing for detailed analysis and comparison across different compression methods.
- **huffman.py**: Provides functions for Huffman encoding and decoding. The module includes mechanisms to build Huffman trees, assign codes based on symbol frequencies, and handle padding for binary data representation [2].
- **bzip2.py**: Implements the Bzip2 compression algorithm by integrating SBWT, MTF encoding, and Huffman coding. This module is optimized for text-heavy datasets and supports efficient file compression and decompression [4].
- **lzw.py**: Implements the Lempel-Ziv-Welch (LZW) algorithm. It dynamically builds a dictionary of repeating patterns in the input data, offering fast and lightweight compression suitable for real-time applications [5].
- **arithmetic.py** and **arithmethic_coder.py**: These modules handle Arithmetic Coding as an alternative entropy coding method. By representing data as fractional ranges based on symbol probabilities, these modules provide finer-grained compression, particularly effective for datasets with skewed symbol distributions [3, 12].

4.2 Compression Pipeline

The compression pipeline integrates multiple stages to achieve optimal efficiency. Each block of data undergoes:

- **Scrambled Burrows-Wheeler Transform (SBWT)**: Reorganizes input data into clusters of similar characters, preparing it for entropy reduction.
- **Move-to-Front (MTF) Encoding**: Reduces data entropy by repositioning frequently used symbols to the front of the list, creating runs of repeated symbols for efficient encoding [6].
- **Run Length Encoding (RLE)**: Applied following the Move-to-Front Encoding, RLE compresses sequences of repeated characters into a single character and count. This method is highly effective for datasets with long sequences of identical data, significantly reducing the data size by simplifying repetitive patterns into compact representations. While particularly powerful in contexts with high data redundancy, its performance may diminish in datasets with high variability [8].
- **Entropy Coding**: The system supports multiple entropy coding methods, including Huffman Coding, LZW, and Arithmetic Coding. Huffman and LZW provide efficient and lightweight compression for structured or repetitive datasets, while Arithmetic Coding offers superior compression ratios for datasets with non-uniform symbol distributions. In particular, Arithmetic Coding dynamically adjusts symbol probabilities during processing, encoding the entire data block as a single fractional range [12]. This method is seamlessly integrated into the pipeline, leveraging modularity for selective use where its benefits outweigh the computational cost.

4.3 Parallel Processing

To handle large datasets efficiently, the system processes data in fixed-size blocks (64 KB by default). Each block is compressed independently, allowing for parallel processing using Python's `concurrent.futures.ProcessPoolExecutor`. During compression, SBWT, MTF encoding, and entropy coding (e.g., Huffman, Arithmetic, or LZW) are applied independently to each block. The modularity of the pipeline enables dynamic selection of the best entropy coding method based on data characteristics.

During decompression, the compressed file is read block by block, and each block is processed in parallel. The system ensures that blocks are recombined in the correct order, restoring the original file without loss of information.

4.4 Key Handling

Key management is central to the security of the system, particularly for SBWT. The Counter Mode (CTR) is used to generate unique sub-keys for each block [9, 10]. A cryptographic hash function, such as SHA-256, combines the user-provided master key with the block number to produce sub-keys. This ensures both CPA security and the ability to process blocks independently.

4.5 Error Handling and Debugging

Robust error-handling mechanisms are incorporated to ensure system stability:

- **Input Validation**: The system verifies the validity of file paths, keys, and data formats before processing.
- **Corrupted Data Detection**: Decompression routines include checks for corrupted or missing blocks to prevent incomplete outputs.
- **Logging**: The system provides comprehensive logging at `INFO`, `DEBUG`, and `ERROR` levels. Debug logs include detailed insights into steps such as frequency table updates, range adjustments in Arithmetic Coding, and block serialization.

4.6 Performance Optimizations

Several optimizations are implemented to enhance performance:

- **Efficient Sorting in SBWT**: Precomputed hashes are used to optimize custom character order generation.
- **SA-IS Algorithm for Suffix Array Construction**: The Suffix Array Induced Sorting (SA-IS) algorithm is employed, achieving a complexity of $O(n \log^2(n))$, which is significantly better than traditional $O(n^2)$ approaches. This improves the efficiency of constructing the index array within the Scrambled Burrows-Wheeler Transform (SBWT).
- **MTF Encoding Efficiency**: List operations are minimized to reduce computational overhead.
- **Arithmetic Coding Precision**: The use of a 32-bit state balances precision and speed, avoiding excessive memory consumption while maintaining compression accuracy.
- **Parallelism**: Block-based processing ensures linear scalability with the number of CPU cores.

The proposed compression system is implemented in Python, leveraging modularity and parallelism to achieve scalability and efficiency. This section describes the key components and their functionalities, outlining their integration within the compression pipeline.

5 Experimental Results

This section presents the results of the experimental evaluation of the proposed compression system. The experiments focus on evaluating the compression ratio, execution time, and scalability of the system. Additionally, a comparative analysis between the coding algorithms used is provided to highlight their respective strengths and trade-offs.

5.1 Case Study: Lorem_Ipsum And Robida

5.1.1 Performance Metrics: The evaluation metrics included:

- **Compression Ratio:** Measures the effectiveness of the compression by comparing the size of the compressed file to the original file, expressed as a percentage. This metric is essential for understanding the reduction in file size achieved by our system [13].

- **Execution Time:** Gauges the speed of both compression and decompression processes. This metric is crucial for assessing the efficiency of the system under different load conditions [12].

- **Scalability:** Assesses the system's capability to handle increasing file sizes without significant performance degradation. This metric is vital for determining the system's practicality in real-world scenarios where data volumes can vary substantially [14].

5.1.2 Results and Analysis This section presents the results of the experimental evaluation of the compression system, focusing on the compression ratio, execution time, scalability, and a comparative analysis of Huffman and arithmetic coding methods.
The system demonstrated competitive compression ratios across all the tested datasets. Huffman coding achieved slightly lower compression ratios compared to arithmetic coding, particularly for datasets with highly non-uniform symbol distributions. On average, arithmetic coding provided a 5–10% improvement in compression ratio over Huffman coding, as shown in Table 1 [3, 12].

5.1.3 Execution Time: Parallel processing significantly improved execution times for large files. Huffman coding was faster due to its simpler algorithmic structure, whereas arithmetic coding incurred additional overhead from precision management. Despite this, the scalability of the system allowed arithmetic coding to handle large datasets efficiently, as shown in Table 2.

5.1.4 Scalability: The block-based processing design ensured that the system scaled effectively with file size. Larger files were compressed in parallel blocks without significant increases in memory usage, and the performance scaled linearly with the number of CPU cores. This scalability confirms the system's suitability for modern multi-core processing environments.

5.1.5 Comparative Analysis: The choice among Huffman, Bzip2, and LZW coding depends on the application and specific requirements. Arithmetic coding stands out in scenarios where maximum compression ratios are needed, especially for datasets with highly non-uniform distributions [3]. However, it incurs higher computational overhead, making it less suitable for real-time or low-latency applications. These trade-offs highlight the flexibility of the system to adapt to different use cases by leveraging the strengths of each algorithm.

File	Mode	Compressed Size (KB)	Compression Ratio (%)
Lorem_Ipsum.txt	Huffman	66	71.38
	Bzip2	46	80.14
	LZW	109	52.70
	Arithmetic	62	73.31
robida.txt	Huffman	2800	70.93
	Bzip2	2400	74.51
	LZW	4900	49.19
	Arithmetic	2500	74.13

Table 1: Compression performance for various files and methods, comparing Huffman, Bzip2, LZW, and Arithmetic coding.

File	Mode	Compression Time (s)	Decompression Time (s)
Lorem_Ipsum.txt	Huffman	0.52	0.10
	Bzip2	0.09	0.09
	LZW	0.52	0.10
	Arithmetic	0.75	0.55
robida.txt	Huffman	9.64	0.54
	Bzip2	0.16	0.12
	LZW	9.80	0.44
	Arithmetic	15.03	10.31

Table 2: Compression and decompression times for various files and methods, comparing Huffman, Bzip2, LZW, and Arithmetic coding.

5.1.6 Limitations While the system demonstrates significant advantages in terms of compression efficiency, scalability, and security, it is not without limitations. Arithmetic coding, though highly efficient in compression, introduces computational overhead due to precision management. Future work could focus on optimizing its implementation or integrating predictive models to dynamically select the most suitable algorithm based on dataset characteristics [14].

5.2 Case Study: Mono-dimensional Dataset

This case study evaluates the performance of the compression methods on a collection of mono-dimensional files selected from well-known datasets. The datasets included in this evaluation are:

- **artifical_corpus**
- **calagry_corpus (large)**
- **canterbury_corpus (large)**
- **protein**

- **silesia**

The test set consists of 38 files with a total size of approximately 2.9 MB. Each file was subjected to 152 tests over a duration of roughly 30 minutes. All datasets were sourced from data-compression.info and affiliated universities.

5.2.1 Performance Metrics The evaluation metrics for this case study include the average compression ratio (%) and the average compression time (seconds). The results are summarized in Table 3 and Figure **??**.

Mode	Avg Compression (%)	Avg Time (s)
LZW	38.70	2.05
Huffman	59.19	2.02
Arithmetic	62.68	2.77
Bzip2	68.37	0.21

Table 3: Average compression ratio and time for the multidimensional dataset.

5.2.2 Results and Analysis The performance across the compression methods showed distinct trends:

- **Bzip2** emerged as the most efficient method, achieving the highest average compression ratio (68.37%) and the fastest average time (0.21 seconds). This makes it the most suitable choice for compressing mono-dimensional datasets with repetitive patterns.
- **Arithmetic Coding** delivered the second-highest compression ratio (62.68%) but had the longest average compression time (2.77 seconds), reflecting the computational overhead associated with its precision-based calculations.
- **Huffman Coding** achieved a moderate compression ratio (59.19%) with an average time of 2.02 seconds, offering a balanced approach for datasets without high variability.
- **LZW** demonstrated the lowest compression ratio (38.70%) and an average time of 2.05 seconds, making it less effective for this dataset.

The results are visualized in Figure **??**, showing the trade-offs between compression ratio and execution time.

5.3 Case Study: Plain Text Wikipedia Dataset

This case study focuses on the performance of our compression system when applied to a large-scale plain text Wikipedia dataset obtained from Kaggle [15]. The dataset comprises a diverse range of articles, providing a robust test bed for evaluating the effectiveness of different compression methods under real-world conditions.

5.3.1 Experimental Setup: The benchmarking process spanned 36 hours, during which we conducted a series of tests to assess the compression ratio, execution time, and scalability of the system across various compression methods: Huffman Coding, Arithmetic Coding, Bzip, and LZW.

5.3.2 Performance Metrics: The evaluation metrics for this dataset included:

- **Number of Successful and Failed Tests:** Captures the reliability of each compression method in handling diverse data content.
- **Average Compression Time per Mode (seconds):** Measures the time efficiency of each compression algorithm.
- **Average Compression Ratio per Mode:** Indicates the effectiveness of each method in reducing file size.

5.3.3 Results and Analysis The system processed a substantial volume of data, revealing several key insights:

- All methods achieved a 100% success rate across tests, showcasing the robustness and reliability of the system when applied to a diverse dataset.
- **Bzip2** demonstrated the fastest processing time, with an average time of 0.81 seconds, while achieving the highest compression ratio at 67.02%. This makes Bzip2 the most efficient method overall, providing a strong balance between speed and compression effectiveness.
- **Huffman Coding** and **LZW** offered moderate compression ratios of 57.69% and 27.92%, respectively, with relatively fast processing times (45.45 seconds for Huffman and 44.30 seconds for LZW). These methods are well-suited for scenarios prioritizing speed over maximum compression.
- **Arithmetic Coding** achieved the second-highest compression ratio at 60.35%, demonstrating its effectiveness with dense informational content. However, it required the most time, averaging 123.98 seconds per test, which makes it less suitable for time-critical applications.

The performance metrics were captured from our benchmark results, and a summary is provided in Table 4.

Compression Mode	Success Rate (%)	Average Time (s)	Average Compression Ratio
Huffman	100	45.45	57.69
Arithmetic	100	123.98	60.35
Bzip2	100	0.81	67.02
LZW	100	44.30	27.92

Table 4: Benchmarking results for the Wikipedia Plain Text dataset with various compression modes.

5.3.4 Comparative Analysis: Based on the results, the choice of compression method depends heavily on the specific requirements of the application:

- **Bzip2** stands out as the optimal choice for a balanced approach, offering the best compression ratio and the fastest processing time, making it highly suitable for mixed data types found in Wikipedia articles.
- **Arithmetic Coding** is ideal for applications where maximizing compression and security is a priority, particularly for dense or complex datasets. However, its higher computational cost makes it less practical for real-time use.
- **Huffman Coding** and **LZW** are better suited for time-sensitive tasks due to their relatively quick processing times, but their lower compression ratios make them less effective for highly redundant or complex textual data.

5.3.5 Limitations Despite the overall high success rates and strong performance of the compression methods, certain limitations were observed:

- **Arithmetic Coding**, while achieving one of the highest compression ratios (60.35%), required significantly more computational time, with an average processing time of 123.98 seconds. This makes it less practical for real-time or large-scale applications where time efficiency is critical.
- **LZW** and **Huffman Coding**, though faster (44.30 and 45.45 seconds, respectively), were less effective in achieving high compression ratios, particularly with datasets containing highly redundant or complex data. Huffman reached a compression ratio of 57.69%, while LZW only achieved 27.92%.
- While **Bzip2** offered the best overall balance with the highest compression ratio (67.02%) and the fastest time (0.81 seconds), it may not perform as well on datasets that deviate significantly from the characteristics of the Wikipedia dataset, requiring further testing in varied scenarios.

6 Discussion

The results of this study highlight the strengths and trade-offs of the proposed compression system compared to traditional methods like Huffman, Bzip2, and LZW. The integration of the Scrambled Burrows-Wheeler Transform (SBWT) with entropy coding techniques demonstrates significant potential in achieving efficient compression while enhancing data security through key-based transformations. However, the findings also point to areas where further optimization and integration of additional algorithms could enhance the system's adaptability and efficiency.

6.1 Compression Efficiency Across Case Studies

The two case studies reveal important insights into the performance of different compression methods under varying dataset characteristics. While Arithmetic Coding excelled in the first case study due to its superior compression ratios for highly non-uniform symbol distributions, Bzip2 emerged as the clear winner in the second case study, demonstrating a remarkable balance between compression ratio and execution time.

6.1.1 First Case Study: Lorem Ipstum/Robida In the first case study, Arithmetic Coding consistently outperformed other methods in terms of compression ratio, achieving up to a 5–10% improvement over Huffman Coding. This performance was particularly pronounced for datasets with non-uniform symbol distributions, where Arithmetic Coding's precision in representing probabilities provided a significant advantage. However, this came at the cost of higher execution times, making it less suitable for real-time applications.

6.1.2 Second Case Study: Mono-dimensional Dataset This case study analyzed the performance of compression methods on a collection of mono-dimensional datasets, including the artificial corpus, calgary corpus, canterbury corpus, protein, and silesia. The test set comprised 38 files with a total size of approximately 2.9 MB on average, sourced from data-compression.info and affiliated universities. Bzip2 achieved the highest compression ratio (68.37%) and the fastest average time (0.21 seconds), making it the best choice for repetitive patterns. Arithmetic Coding followed with a strong compression ratio (62.68%) but exhibited the longest execution time (2.77 seconds), reflecting its computational complexity. Huffman Coding provided a balanced approach with a moderate compression ratio (59.19%) and an average time of 2.02 seconds. LZW showed the lowest compression ratio (38.70%) and a slightly longer average time (2.05 seconds). These results highlight that Bzip2 is optimal for mono-dimensional datasets requiring efficiency and speed, while Arithmetic Coding suits scenarios prioritizing compression precision over execution time.

6.1.3 Third Case Study: Plain Text Wikipedia Dataset In the third case study, Bzip2 demonstrated superior performance, achieving the highest compression ratio (67.02%) while also maintaining the fastest average execution time (0.81 seconds). The combination of the Burrows-Wheeler Transform (BWT), Move-to-Front (MTF) and Run-Length (RLE) encodings, allowed Bzip2 to handle the diverse and repetitive patterns in the Wikipedia dataset efficiently. Arithmetic Coding, while still effective, was less practical in this scenario due to its higher computational overhead and longer execution times.

These findings highlight the importance of selecting a compression method tailored to the specific characteristics of the dataset. Arithmetic Coding excels in scenarios requiring maximum compression for dense or skewed data, while Bzip2 is better suited for general-purpose applications requiring a balance between speed and compression efficiency.

6.1.4 Visual Comparison of Results The differences between the two case studies are further illustrated in Figure ??. The chart compares the average compression ratios and execution times of the methods, emphasizing the trade-offs between speed and compression efficiency.

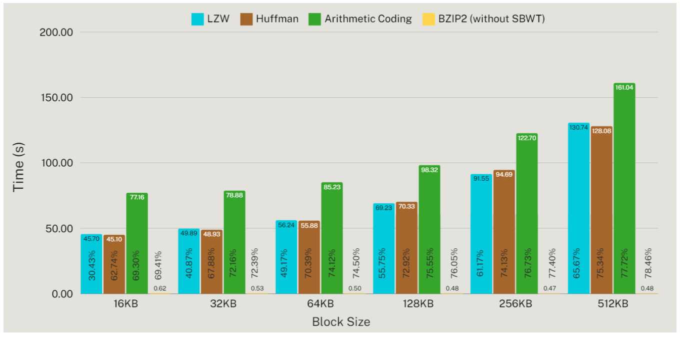

Figure 1: Comparison of compression ratios across Huffman, Bzip2, LZW, and Arithmetic Coding for the lorem ipsum/robida files.

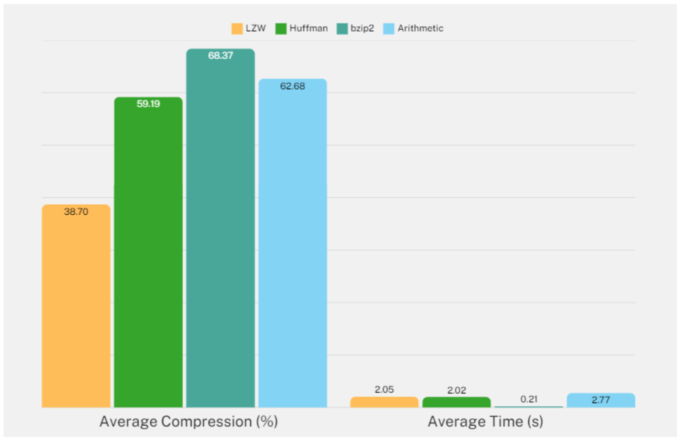

Figure 2: Average compression ratio and time across LZW, Huffman, Arithmetic, and Bzip2 for the multidimensional dataset.

In summary:

- In the first case study, the dataset's highly non-uniform distributions made Arithmetic Coding the ideal choice, delivering the best compression ratios despite its slower processing times.

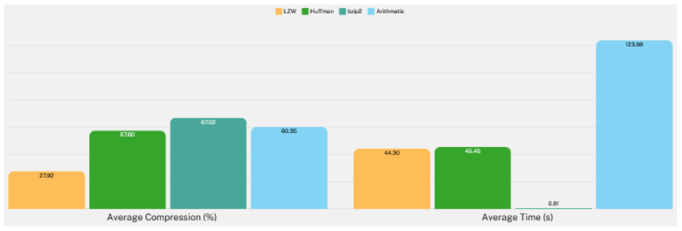

Figure 3: Comparison of compression ratios across Huffman, Bzip2, LZW, and Arithmetic Coding for the wikipedia dataset.

- In the second case study, the diversity and repetition in the Wikipedia dataset favored Bzip2, which offered both speed and high compression efficiency.

This comparative analysis underscores the adaptability of the proposed compression system, which leverages the strengths of different algorithms to optimize performance based on dataset characteristics.

6.2 Scalability and Parallelism

The block-based processing design of the system ensured effective scalability across varying file sizes. Parallel processing allowed each block to be independently compressed and decompressed, distributing the computational workload across multiple CPU cores. This scalability mitigated the computational overhead associated with Arithmetic Coding, making it viable for large-scale datasets. Nevertheless, the higher complexity of Arithmetic Coding required careful optimization to fully capitalize on the benefits of parallel processing.

6.3 Security Considerations

The use of the SBWT with a key-driven scrambling mechanism added an additional layer of data obfuscation [11]. The Counter Mode (CTR) employed for key management ensured CPA (Chosen Plaintext Attack) security by deriving unique, deterministic sub-keys for each block. This mechanism not only enhanced data security but also facilitated independent processing of blocks during compression and decompression.

6.4 Trade-offs and Limitations

While the system exhibits flexibility and high compression performance, certain limitations remain:

- **Arithmetic Coding Overhead:** The computational cost associated with managing ranges and probabilities in Arithmetic Coding makes it less suitable for real-time applications. Further optimization of its implementation could mitigate this limitation.

- **Algorithm Selection:** While the modular design allows the selection of different entropy coding techniques, the system does not currently automate the selection process based on dataset characteristics. Integrating predictive models [14] to analyze data properties and recommend the most efficient algorithm would enhance adaptability.
- **Compression-Decompression Symmetry:** The need to serialize metadata such as symbol tables and probability distributions introduces minor overhead during decompression. Future work could explore more compact serialization formats to minimize this impact.

7 Future Directions

Future improvements to the system could include a combination of strategic enhancements aimed at improving adaptability, efficiency, and security:

- **Predictive Algorithm Selection:** Leveraging machine learning models to analyze data characteristics, such as symbol distributions and redundancy patterns, to dynamically recommend the optimal compression technique for each dataset. Such predictive models would automate the decision-making process, ensuring both speed and efficiency [14].
- **Hardware Acceleration:** Exploring GPU-based implementations or custom hardware solutions (e.g., FPGAs) to reduce the computational overhead of Arithmetic Coding and other high-complexity algorithms. These solutions could make the system viable for real-time applications and large-scale datasets [16].
- **Enhanced Security Features:** Investigating additional cryptographic methods to complement the key-driven scrambling in SBWT. Techniques such as homomorphic encryption or lightweight encryption algorithms could ensure data confidentiality during storage and transmission without significantly affecting compression efficiency [10].
- **Enhanced Metadata Optimization:** Refining serialization formats for metadata, such as frequency tables and probability models, to reduce decompression overhead. Future approaches could explore adaptive models to minimize the need for explicitly storing metadata.
- **Parallelism Beyond Blocks:** While block-based parallelism ensures scalability, exploring intra-block parallelism could further optimize performance. Multi-threaded processing of frequency tables or entropy coding within a single block could significantly enhance throughput for large datasets.
- **Hybrid Compression Pipelines:** Developing hybrid pipelines that combine multiple compression techniques for datasets with varying characteristics. For example, applying Arithmetic Coding to high-entropy segments while using Huffman or LZW for low-entropy regions could optimize both speed and compression ratio [2, 5].
- **Adaptability to Non-Text Data:** Extending the system's applicability to non-text datasets, such as images or videos, by integrating specialized pre-processing steps like quantization or transform coding. This would make the system versatile across multiple domains.
- **Evaluation on Real-World Datasets:** Expanding experimental validation to include diverse real-world datasets, such as genomic data, multimedia files, and sensor logs. This evaluation would help identify domain-specific optimizations and inform further refinements [13].

These advancements would enable the system to dynamically adapt to a broader range of datasets and use cases, achieving an optimal balance between compression efficiency, speed, scalability, and security. By addressing these future directions, the system has the potential to become a robust and versatile solution for modern data compression challenges.

8 Conclusions

This study presented a modular and scalable compression system that integrates the Scrambled Burrows-Wheeler Transform (SBWT), Move-to-Front (MTF) encoding, and multiple entropy coding techniques, including Huffman, Bzip2, LZW, and Arithmetic Coding. The proposed system demonstrates significant advantages in terms of compression efficiency, scalability, and security.

8.1 Key Findings

The experimental evaluation highlighted the following:

- **Compression Efficiency:** Arithmetic Coding achieved the highest compression ratios, particularly for datasets with highly skewed symbol distributions, outperforming Huffman coding by 5–10% [3, 12]. Bzip2 [4] and LZW [5] were effective in handling repetitive datasets but with trade-offs between compression ratio and execution time.
- **Execution Time:** Huffman coding demonstrated the fastest processing times, while Arithmetic Coding incurred additional overhead due to precision management. Parallel processing mitigated these computational demands, enabling scalability for larger datasets.
- **Security:** The SBWT, enhanced with key-driven scrambling and managed using Counter Mode (CTR), added a robust layer of data obfuscation, ensuring CPA security and allowing block-wise independent processing.

8.2 Contributions

The key contributions of this work include:

- A modular compression framework capable of integrating and evaluating multiple entropy coding techniques.
- The successful integration of Arithmetic Coding into a scalable system, balancing its superior compression efficiency with computational challenges.
- Enhanced security through key-driven scrambling mechanisms in SBWT, ensuring data obfuscation without compromising performance.
- Scalability achieved through block-based parallel processing, demonstrating linear performance improvements with increasing CPU cores.

In summary, the proposed system represents a significant step forward in achieving efficient, secure, and scalable data compression. By addressing the identified limitations and implementing the future directions discussed previously, this system has the potential to adapt dynamically to diverse datasets and real-world applications, achieving an optimal balance between compression efficiency, speed, scalability, and security.main

References

[1] Michael Burrows and David J. Wheeler. 1994. A block-sorting lossless data compression algorithm. In *Technical Report 124, Digital Equipment Corporation.*

[2] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE.*

[3] Jorma Rissanen and G. G. Langdon. 1979. Arithmetic coding. *IBM Journal of Research and Development.*

[4] Julian Seward. [n. d.] The bzip2 and libbzip2 official home page. http://www.bzip.org. Accessed: 2024-12-10. ().

[5] Jacob Ziv and Abraham Lempel. 1978. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory.*

[6] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. 1986. A locally adaptive data compression scheme. *Communications of the ACM.*

[7] Author Names. 2020. Selective run-length encoding. *arXiv*. https://arxiv.org/abs/2312.17024.

[8] S. Sarika et al. 2013. Improved run length encoding scheme for efficient compression. *International Journal of Engineering Research and Applications*, 3, 2017–2020, 6. http://www.ijera.com.

[9] Morris Dworkin. 2001. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Technical report SP 800-38A. NIST.

[10] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM.*

[11] M. Oğuzhan Külekci. 2012. On scrambling the burrows–wheeler transform to provide privacy in lossless compression. *Computers Security*. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2011.11.005. https://www.sciencedirect.com/science/article/pii/S0167404811001416.

[12] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. *Communications of the ACM.*

[13] David Salomon. 2007. *Data Compression: The Complete Reference.* (4th edition). Springer.

[14] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[15] Ahrmadi. 2023. Plain text wikipedia dataset. Accessed: January 2025. (2023). https://www.kaggle.com/datasets/ahrmadi/plain-text-wikipedia-2023.

[16] Nvidia Corporation. 2024. Cuda toolkit documentation. (2024). https://docs.nvidia.com/cuda/.