FTCG: Fine-to-Coarse Multi-Granularity Grouping for Migratory Compression

LiZhi Zhang[1,†], XiangLong Shi[1,†], Fang Zou[1,*]
{546547814@qq.com, shi3329163446@gmail.com, zoudafang2002@gmail.com}
School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China[1]
[†]These authors contributed equally. [*]Corresponding author.

Abstract. With the exponential growth of data volume, lossless compression technology has become a crucial technique for alleviating storage pressure. Existing Migratory Compression (MC) algorithms exhibit inefficient utilization of the compression sliding window due to excessively large groups and distant similar data blocks. To address this issue, this paper proposes FTCG, a Fine-to-Coarse Multi-Granularity grouping migratory compression scheme. FTCG employs Hierarchical Super-Features (HSF) to achieve progressive similarity quantification from fine to coarse granularity, thereby prioritizing the aggregation of highly similar data blocks to enhance local redundancy elimination. Simultaneously, an efficient sorting-based grouping strategy, HSFRank, is designed to replace the traditional greedy algorithm with linear complexity, significantly reducing computational overhead. Our evaluation results show that FTCG improves the average compression ratio by 42.50% across nine datasets, with compression ratio gains of 58.20% and 51.57% on the Bash and LKT datasets, respectively, and a maximum throughput improvement of 60.60%.

Keywords: Lossless Compression, Migratory Compression, Super-Features, Data Block Grouping, Large-scale Storage Systems

1 Introduction

With the popularization of big data and cloud computing technologies, global data volume is growing explosively. According to IDC (International Data Corporation) predictions, the total global data volume will exceed 175 ZiB by 2025, with enterprise storage costs accounting for over 40% of total IT expenditure[1]. In this context, lossless compression technology [2] has become an integral component of the storage stack. By eliminating data redundancy, it enables efficient storage utilization and has been widely deployed in databases, backup systems, and distributed storage. Modern lossless compression algorithms like LZ4[3] and Zstandard[4] have achieved a significant balance between compression ratio and speed by optimizing dictionary matching and entropy coding.

IIKI 2025, December 18-21, Chengdu, People's Republic of China

DOI 10.4108/eai.18-12-2025.2365285

Most lossless compression algorithms operate by identifying and exploiting redundant data within a sliding window. For example, the DEFLATE algorithm used by gzip[5] employs a 32 KiB compression sliding window, while the LZ4 algorithm employs a 64 KiB compression sliding window. Although employing a larger compression sliding window increases the probability of detecting duplicate data, it also incurs a significant performance penalty. Therefore, to maximize the redundancy removal benefits of the compression window, optimization schemes for data reordering have been proposed: grouping identical or similar strings together before compression to improve compression performance. A seminal example is the Burrows-Wheeler Transform (BWT)[6], which reorders strings within a data block. If a string has recurring substrings, the same letters will be adjacent after sorting. Scaling this concept, Xing Lin et al. proposed a large-scale (typically tens of GiB or larger) coarse-grained BWT called Migratory Compression (MC)[7]. MC logically divides data into blocks, computes features for each data block, and then groups the similar data blocks together, enabling standard compressors to find repeated strings in adjacent blocks. Moreover, MC acts as a preprocessor and can be combined with any adaptive lossless compressor (e.g., gzip, bzip2, or 7zip)[8].

However, MC often results in single groups that far exceed the compression window size. Furthermore, the lack of finer-grained similarity differentiation within the same group causes some highly similar data blocks to be too far apart within the group, unable to benefit from the deduplication capabilities of the compression window. To address these issues, we proposes FTCG, a Fine-to-Coarse Multi-Granularity Grouping migratory compression scheme. Its key innovations and contributions are:

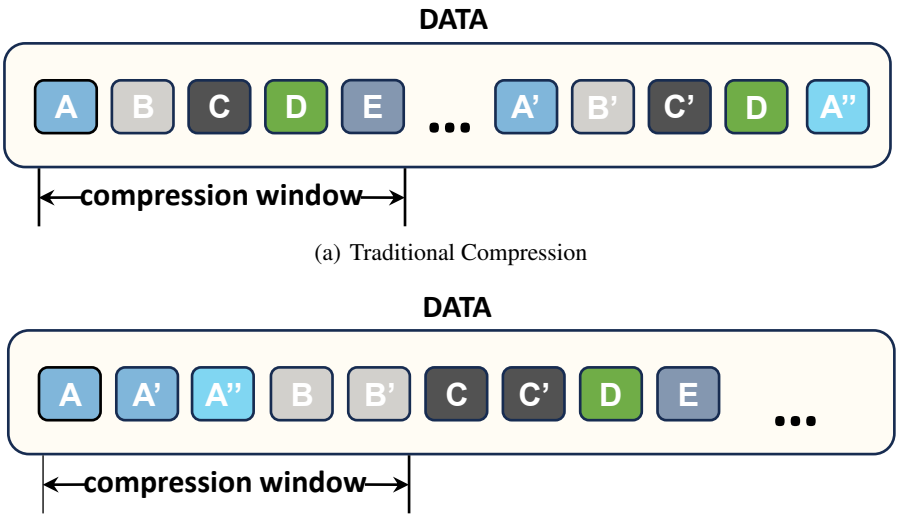
- Multi-granularity feature fusion: Through Hierarchical Super-Features, progressive quantification of similarity from fine-grained to coarse-grained is achieved, prioritizing the aggregation of highly similar data blocks.
- Efficient sorting-based grouping: The original group merging algorithm is improved, replacing $O(n^2)$ greedy merging with $O(n)$ sorting complexity.

2 Background and Related Works

2.1 Lossless Compression and Migratory Compression

Lossless compression reduces storage footprint by eliminating data redundancy while preserving data integrity. Typical lossless methods include dictionary-based LZ series algorithms (e.g., LZ77, LZ78)[9] and entropy coding methods based on statistical models (e.g., Huffman coding[10]). This work focuses on dictionary-based compression algorithms. Dictionary-based compression algorithms achieve data reduction by finding redundant data within a compression sliding window. To maximize the redundancy removal benefit of the compression window, BWT was proposed. The core idea of BWT is to reorder the string to be compressed. If the string has multiple identical substrings, the same substrings will be adjacent after sorting. While BWT operates at the byte level, and due the demand for efficient compression of massive-scale data, a larger-scale and coarse-grained BWT called MC was proposed. MC divides data into blocks based on certain logic, and these blocks

can be fixed or variable length[11]. It then computes features for each data block, stored as a fixed-length string. Data blocks sharing an identical feature are guaranteed to have identical or similar content. Finally, blocks are reordered and grouped based on features before compression. In summary, MC places similar content in adjacent positions from a macro perspective to improve data utilization within the compression sliding window.



(a) Traditional Compression

(b) Migratory Compression

Fig. 1. Comparison of Traditional Compression and Migratory Compression.

Fig. 1 compares traditional compression and MC. Consider blocks A' and A'' at the end of the data stream, which are similar but not identical to block A at the beginning. When using traditional compression, the compressor searches for similar content within a limited window, so the blocks A' and A'' in the data are not compressed relative to block A. When using MC, these similar blocks are clustered, which also includes two other similar blocks B and B'. Note that for two completely identical blocks D, MC only keeps an index for the second block relative to the first and does not put it into the compressor.

2.2 Similarity Detection and Data Blocks Grouping

In grouping migratory compression, the greedy algorithm is an intuitive strategy that seeks the maximum compression ratio gain at each step for a globally optimal result. However, it is also foreseeable that the greedy strategy incurs substantial computational overhead. For large-scale applications where computational efficiency is critical, the feature value method has proven more feasible and is consequently employed in grouping migratory compression. Similarity detection method[12] refer to comparing and evaluating the similarity between data blocks by calculating feature values. Detection accuracy reflects the degree of approximation between actual similarity

and estimated similarity. Odess[13] is a scheme with relatively excellent compression performance and effectiveness among similarity detection methods. This section will introduce the Odess feature extraction method; improvements to Odess's feature extraction method will be described in detail in Section 4.2. The Odess feature extraction method follows this three step:

- Characterizing: In this step, Odess employs a rolling window to segment a data block and concurrently uses the Gear hash[14] to compute a series of fingerprints for each segment. For example, if the data block size is 8 KiB and the rolling window size is 32 bytes, the total number of sliding windows is 256. The fingerprints for each window is: fp_1, fp_2, ..., fp_{255}, fp_{256}.
- Selecting: In this step, Odess employs an extracting method to reduce the scale of the features. It checks each window's hash value fp_i against a predefined condition (e.g., $fp\&mask == 0$). Fingerprints satisfying the condition are selected into the proxy set. The sampling rate is determined by the mask (e.g., when $mask = 0x1c$ (binary 11100), the sampling rate is 1/8). For each selected fp_i, it uses twelve sets of parameters (a_r, b_r) to perform linear transformation $f_{ir} = a_r fp_i + b_r$. For each set of parameters (a_r, b_r), select the minimum value of f_{ir} as the original feature F_r for this set of parameters. Finally, 12 original features $\{F_r\}$ were obtained.
- Indexing: In this step, features are grouped into several Super-Features [15]. For example, given 12 features $F_1, F_2, \ldots, F_{12}$, they can be grouped into three Super-Features as follows:
  - $SF_1 = \text{hash}(F_1, F_2, F_3, F_4)$
  - $SF_2 = \text{hash}(F_5, F_6, F_7, F_8)$
  - $SF_3 = \text{hash}(F_9, F_{10}, F_{11}, F_{12})$

  Then it generates an key-value indexing that associates super-features with their corresponding data blocks. Furthermore, we can only generate SF_1 for each data blocks, and only maintain $SF_1 - Table$, to aviod the computational overhead from hash table set difference operations in data blocks grouping process, the specific reasons and details will be explained in Section 3.1.

3 Motivation

3.1 Shortcomings of Migratory Compression

Our processing aims to maximize the concentration of highly similar data blocks within a single compression sliding window. But the grouping process in MC, which relies on Super-Features, still suffers from several key shortcomings. We will detail these problems below and substantiate our analysis with experimental results and theoretical examination.

Data Layout Inefficiency:

As shown in Fig. 2, the compression sliding window can only contain five data blocks, A_0 to A_4. However, block A_0 and block A_6 are the most similar pair within this group, and placing them within the same compression window would yield better compression effects, but MC cannot identify this. Then we will demonstrate them through experiments.

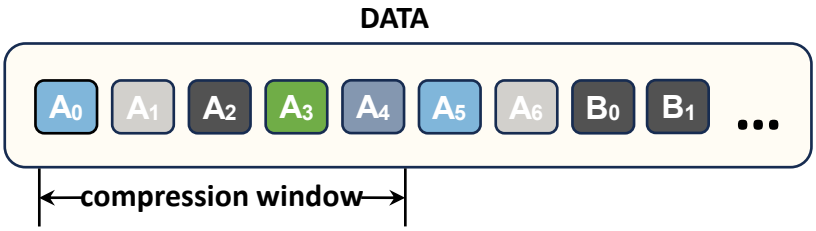
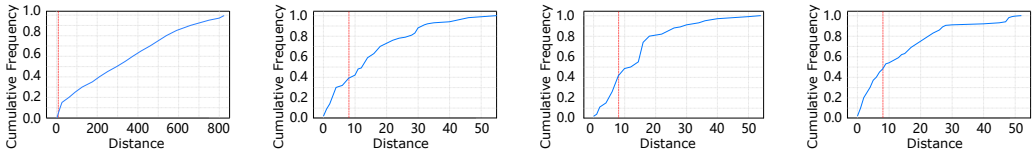


Fig. 2. Data Layout Inefficiency.

The experiments measured the cumulative frequency distribution of the distance between each data block and its most similar block within the largest group obtained by the Odess across different datasets(e.g., if the i-th block is most similar to the j-th block, the distance is $|i-j|$), see dataset's details in Section 5.1.



(a) Chromium (b) Cassandra (c) Bash (d) Glibc

Fig. 3. Distribution of Data Block Similarity within Single Groups Across Datasets.

As shown in Fig. 3, the red vertical line indicates the maximum distance the compression sliding window can contain. LZ4 is uniformly used as the compression algorithm in all experiments. A fixed-length data chunking method is adopted, with a chunk length of 8KiB. Since LZ4's compression sliding window size is 64KiB, the group upper limit is set to 8 data blocks.

The experiments show that after the grouping process of MC, in small datasets like bash, Cassandra, and glibc, over 50% of the data blocks have the highest degree of similarity, but the distance between them within the same group exceeds the size of compression sliding window, preventing them from benefiting from the redundancy removal offered by the compression sliding window. This phenomenon exceeds 90% in large datasets like Chromium.

A Major Performance Bottleneck Arises From Hash Table Set Difference Operations:

The specific feature calculation process of Odess has been mentioned in Section 2.2. We found that after generating super-feature values for each data block using Odess's feature extraction method, in the index, each block has three corresponding super-feature values SF_1, SF_2, SF_3. However, when using the SF_1 for group compression, it is difficult to subsequently use the SF_2 and SF_3 for further grouping. This is because applying the other two super-features for grouping would disrupt the existing groups, invalidating all groupings made by the SF_1. Of course, we could group only the isolated blocks in the SF_1 based on the other two super-features, but since there is no containment relationship between the super-feature values, this would involve set difference operations

on indexes, which has high computational complexity and is unsuitable for practical application.

The above analysis offers an important guidance for the subsequent development of enhanced algorithms, the guidance is: **our grouping migratory compression strategy should enhance similarity detection accuracy and avoid hash table set difference operations.**

3.2 Performance Bottleneck of the Greedy Algorithm

As outlined in Section 2.2, the greedy algorithm incurs prohibitively high computational overhead. Nonetheless, to pursue higher compression efficacy, its underlying principle remains instructive for the grouping process. Therefore, this section provides a detailed analysis of its specific procedural steps. The specific process of the greedy algorithm's grouping strategy is as follows:

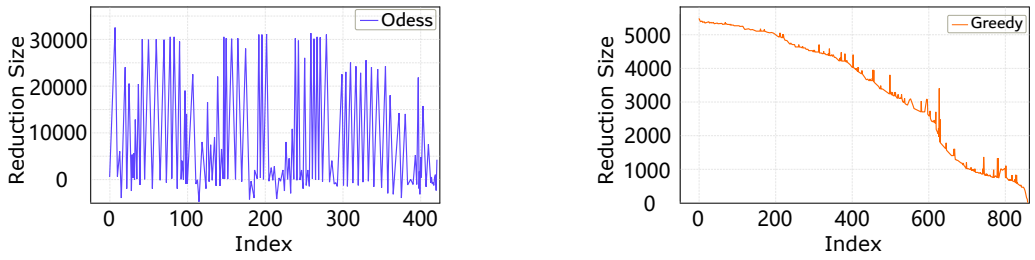
- Step1: Input data blocks: Input n data blocks into the group compression system;
- Step2: Initial compression: Perform lossless compression on each data block individually to obtain the original compressed size C_i $(0 < i \leq n)$;
- Step3: Attempt merging: Attempt to merge every two units i, j (a unit can be a data block or a merged but not full group) and perform lossless compression. The Reduction Size is obtained by subtracting the original two compressed sizes C_i and C_j from the merged compressed size C_{merged} and recorded in a table;
- Step4: Select optimal merge: Merge the two units with the largest Reduction Size greater than 0 into a new unit;
- Step5: Repeat Step 3 and Step 4: During repetition, a full traversal is not needed each time; instead, compress every newly generated unit losslessly against all existing units, compute the Reduction Size, and record the result in the table.

The time complexity of the first grouping is $O(n^2)$, as it requires pairwise merge attempts for all n data blocks. In subsequent grouping processes, only the newly generated block needs to attempt merge against all existing units each time, so the time complexity per step is $O(n)$. The total time complexity is $O(n^2)$. Taking the grouping and compression of 1024 data blocks as an example, compares the compression effects and performance of the feature value method and the greedy algorithm, as shown in Table 1.

Method	**Processing Time (s)**	**Compression Ratio**
Feature Value Method	1	1.61
Greedy Algorithm	505	1.74

Table 1: *Performance Comparison Between Feature Value Method and Greedy Algorithm*

Fig. 4 shows the Reduction Size benefit brought by each step in the Odess and greedy grouping processes. Observing the grouping process of the greedy algorithm and the feature value method



(a) Reduce Size Line Chart for Odess

(b) Reduce Size Line Chart for Greedy

Fig. 4. Analysis of Odess and Greedy Algorithm Grouping Processes.

from the perspective of additional redundancy removal, the horizontal axis represents the number of data blocks that have been grouped, and the vertical axis represents the additional redundancy removal benefit whenever a data block is grouped. We can compare their grouping processes. In the feature value method's grouping process, it can only merge blocks with the same feature value each time. Since feature values of the same granularity cannot distinguish similarity levels, the Reduction Size fluctuates significantly. In contrast, the greedy algorithm selects the currently achievable maximum Reduction Size for each merge to obtain the largest additional redundancy removal, so the Reduction Size shows a decreasing trend. What we desire is the trend in the greedy algorithm that each step achieves the maximum possible redundancy elimination, rather than the uncertain and fluctuating redundancy reduction observed at each step of the feature-based method.

For performance considerations, we cannot use the greedy algorithm in practical applications, but its processing order for data blocks of different similarity is worth referencing: from Fig. 4, it can be inferred that during grouping, one can simulate the greedy logic: **prioritize processing blocks with higher similarity, and then use blocks with lower similarity to fill incomplete groups. This strategy would achieve a better compression ratio.**

4 Design

4.1 Design Overview

To achieve efficient grouping migratory compression and overcome the limitations of traditional feature value methods and greedy algorithms, this paper proposes FTCG. The core methods of the FTCG scheme include:

- Hierarchical Super-Feature (HSF) method: First, unlike Odess, which evenly partitions the 12 raw features into three super-features, the original features are divided into three levels of grain: fine-grained, medium-grained, and coarse-grained. Super-feature values, with containment relationships are constructed through concatenation operations. Then data blocks are

merged layer by layer from high similarity SF_c to low similarity, prioritizing tight aggregation of highly similar blocks while avoiding hash table set-difference operations.

- Sorting-based HSF improvement strategy (HSFRank): First, sort the data blocks by super-feature value SF_c, making similar blocks physically adjacent and simplifying the grouping process. Then, based on the sorting results, directly partition into fixed-size groups (e.g., 8 blocks per group), replacing traditional multi-layer merging with a single sorting operation, significantly reducing time complexity.

4.2 HSF: Similarity Partitioning Based on Hierarchical Features

To maximize the advantage of the compression sliding window, we propose a method HSF that can effectively distinguish the similarity level of data blocks.

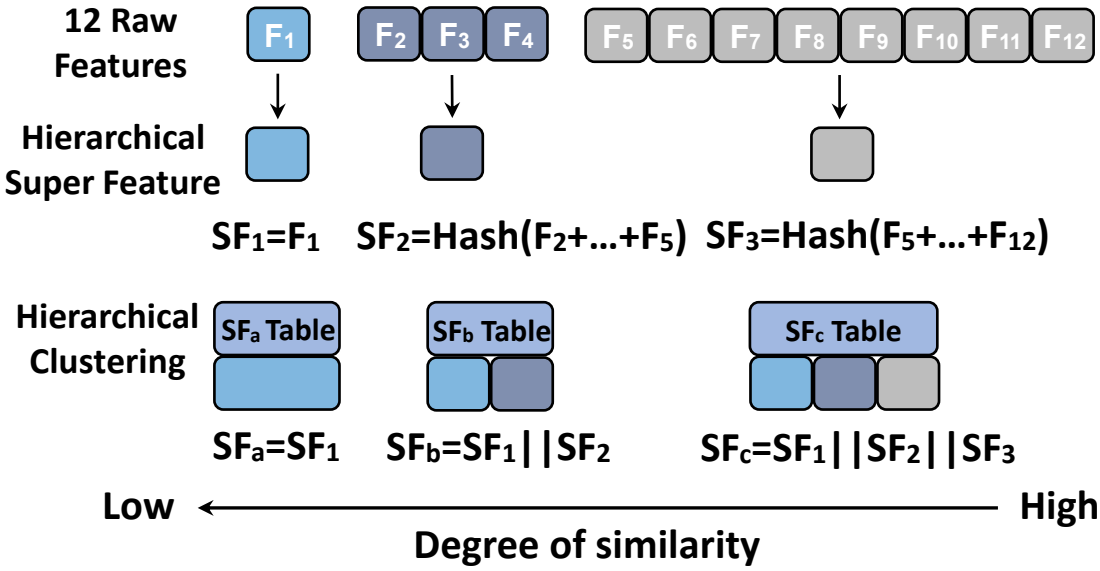


Fig. 5. Hierarchical Feature Value Generation Process

As shown in Fig. 5, HSF utilizes the 12 original features extracted by Odess, first dividing them into 3 groups, containing 1, 3, and 9 original features respectively. Then, through linear transformation, Super-Feature values SF_1, SF_2, SF_3 of different granularities are obtained. At this point, the three Super-Feature values can represent different degree of similarity, but since there is no containment relationship between SF_1, SF_2, and SF_3, grouping based on these three Super-Feature values would still involve hash table set-difference operations.For this reason, we perform further linear transformation and combination on SF_1, SF_2, and SF_3 to make them form a certain containment relationship to avoid hash table set-difference operations. Let $SF_a = SF_1$, $SF_b = SF_1 \| SF_2$, $SF_c = SF_1 \| SF_2 \| SF_3$, where "$\|$" denotes the concatenation operation of feature values, each SF has a

length of 8 B. Now SF_a, SF_b, SF_c are essentially combinations of 1, 4, and 12 features, respectively, with feature value lengths of 8 B, 16 B, and 24 B, and possess a containment relationship from SF_c down to SF_a. This avoids hash table set-difference operations in subsequent group merging.

4.3 HSFRank: Efficient Sorting-Based Grouping Strategy

Although the HSF method solves the similarity measurement problem, it introduces non-negligible merge time overhead. To further optimize performance, we propose the HSFRank method. This algorithm significantly reduces computational overhead by improving data structures and merge strategies:

- Simplify the feature value table structure, retaining only the $SF_c - Table$.
- All data blocks are sorted based on their SF_c values in lexicographical order, causing blocks with similar SF_c values to naturally cluster together.

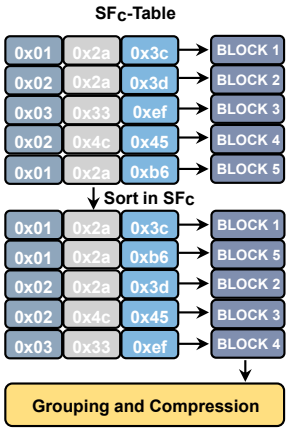


Fig. 6. Schematic Diagram of HSFRank Sorting and Grouping

As shown in Fig. 6, this sorting-based method can replace the traditional two greedy merge processes with a single sorting operation. Specifically, sorted lexicographically based on their SF_c values. Use the first 8 bytes of the sorted data blocks to partition. The individual parts after partitioning do not need to consider the subsequent 16 bytes; directly use fixed-size grouping, with each group containing 8 data blocks. Move to the next part only after grouping is complete for the current part.

The rationale for this is that through sorting preprocessing, blocks with the same SF_c are naturally most adjacent, and blocks with the same SF_b (first 16 bytes) will be secondarily adjacent. Simultaneously, as long as the first 8 bytes are the same (essentially SF_a is the same), it ensures a certain level of similarity among blocks within the group. This means we can make highly similar blocks grouped together first due to adjacency, and generally similar blocks will gradually fill in. This significantly improves processing efficiency while maintaining compression effectiveness.

5 Evaluation

5.1 Experimental Setup

The experiment uses a server with the Debian operating system as the experimental platform. The server is equipped with 32 Intel(R) Xeon(R) E5-2683 processors and memory. This experiment selects 9 datasets of varying scales, as shown in Table 2, whose logical sizes range from 3.48 MiB to 14,627 MiB, spanning three orders of magnitude.

Dataset	**Size (MiB)**	**Dataset**	**Size (MiB)**
Fdisk[16]	3.48	Glibc[17]	196.47
Automake[18]	8.69	LKT[19]	824.50
Coreutils[20]	9.85	Cassandra[21]	1409.00
Smalltalk[22]	45.10	Chromium[23]	14627.00
Bash[24]	116.44		

Table 2: *Experimental Dataset Specifications*

5.2 System Performance Analysis

This section tests the compression ratio improvement and throughput optimization brought by the overall scheme through experiments. The baseline method employs the Odess feature extraction technique and groups data blocks based on a single Super-Features. The HSF and HSFRank methods, detailed in Section 4, are evaluated comparatively.

Dataset	**Baseline**	**HSF**	**HSFRank**	**Improvement**
Cassandra	1.92	2.70	2.70	40.62%
LKT	2.54	3.85	3.85	51.57%
Chromium	2.64	3.67	3.67	39.01%
AutoMake	2.19	2.92	2.92	33.33%
Bash	2.56	4.05	4.05	58.20%
Coreutils	2.98	4.28	4.27	43.62%
Fdisk	2.93	3.80	3.80	29.69%
Glibc	2.90	4.18	4.18	44.13%
Smalltalk	2.62	3.73	3.73	42.37%

Table 3: *Compression Ratio Comparison of Different Schemes and Improvement*

As shown in Table 3, compared to the baseline, HSF significantly improve the compression ratio. The complete scheme significantly improves compression ratio without substantially affecting the compression ratio. The nearly identical compression ratios of HSF and HSFRank methods stem

from their shared feature representation. The key improvement in HSFRank lies in the grouping stage. Therefore, its performance gain is primarily reflected in throughput, not in further improving the compression ratio.

Dataset	Baseline	HSF	HSFRank	Improvement
Cassandra	96.63	100.03	129.06	33.56%
LKT	89.28	114.28	142.90	60.06%
Chromium	133.33	94.38	169.63	27.23%
AutoMake	58.71	70.11	82.60	40.69%
Bash	93.28	144.01	147.18	57.78%
Coreutils	58.69	88.46	91.36	55.61%
Fdisk	30.82	46.61	47.23	53.24%
Glibc	98.40	89.26	151.25	53.71%
Smalltalk	87.16	109.55	139.08	59.57%

Table 4: *Throughput Comparison of Different Schemes (MiB/s) and Improvement*

As shown in Table 4, efficient sorted grouping (HSFRank) significantly reduces computational overhead. In the Chromium and Glibc datasets, the throughput of HSF shows a decrease compared to the baseline. This variation can be attributed to inherent differences across datasets, combined with the fact that HSF employs an approximate greedy algorithm during group aggregation. Consequently, HSFRank demonstrates a steady and universal throughput improvement over both the baseline and HSF across all datasets. It can be applied to many complex real-world datasets, providing a comprehensive solution for real-time compression scenarios.

6 Conclusion and Future Work

This paper identified two critical limitations in existing MC algorithms: insufficient feature quantification accuracy and high computational overhead of grouping. To overcome these challenges, we proposed FTCG, a Fine-to-Coarse Multi-Granularity grouping migratory compression scheme. Our solution makes two key contributions: First, we designed hierarchical super features (HSF) with containment relationships, enabling progressive similarity quantification from fine to coarse granularity. This ensures that highly similar data blocks are prioritized for aggregation. Second, we introduced HSFRank, an efficient sorting-based grouping strategy that replaces the traditional $O(n^2)$ greedy algorithm with a linear-complexity process.

Next, our future work is the impact of FTCG and MC on spatial locality. Looking forward, we must confront the effect of compression-driven reorganization on spatial locality. Although FTCG and MC group similar blocks to boost compression, this global reshuffling scatters the native data sequence, impairing locality and raising access overhead. This divergence creates tension between storage efficiency and read performance. Consequently, a key research avenue is to embed locality preservation into FTCG's framework, aiming to navigate a balance between exceptional compression and responsive data access.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Reinsel D, Gantz J, Rydning J. The Digitization of the World: From Edge to Core. IDC; 2018.

[2] Xia W, Jiang H, Feng D, Douglis F, Shilane P, Hua Y, et al. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. Proceedings of the IEEE. 2016;104(9):1681-710.

[3] Bartík M, Ubik S, Kubalik P. LZ4 compression algorithm on FPGA. In: 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS); 2015. p. 179-82.

[4] Chen J, Daverveldt M, Al-Ars Z. FPGA Acceleration of Zstd Compression Algorithm. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2021. p. 188-91.

[5] Plugariu O, Petrica L, Pirea R, Hobincu R. Hadoop ZedBoard cluster with GZIP compression FPGA acceleration. In: 2019 11th International Conference on Electronics, Computers and Artificial Intelligence (ECAI); 2019. p. 1-5.

[6] Burrows M, Wheeler DJ. A Block-Sorting Lossless Data Compression Algorithm. Digital Equipment Corporation, Systems Research Center; 1994.

[7] Lin X, Lu G, Douglis F, Shilane P, Wallace G. Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility. In: 12th USENIX Conference on File and Storage Technologies (FAST 14). Santa Clara, CA: USENIX Association; 2014. p. 256-73.

[8] Pavlov I. 7-Zip: A File Archiver with a High Compression Ratio;. [Online]. Available from: `https://www.7-zip.org/`.

[9] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory. 1977;23(3):337-43.

[10] Huffman DA. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE. 1952;40(9):1098-101.

[11] Xia W, Zhou Y, Jiang H, Feng D, Hua Y, Hu Y, et al. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association; 2016. p. 101-14.

[12] Broder AZ. On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171); 1997. p. 21-9.

[13] Zou X, Deng C, Xia W, Shilane P, Tan H, Zhang H, et al. Odess: Speeding up Resemblance Detection for Redundancy Elimination by Fast Content-Defined Sampling. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE); 2021. p. 480-91.

[14] Lee J. Gear Hashing for Content-Defined Chunking; 2023. Blog post. Available from: `https://joshleeb.com/posts/gear-hashing.html`.

[15] Purushottam Kulkarni UoM, Douglis F, LaVoie J, Tracey JM. Redundancy Elimination Within Large Collections of Files. In: 2004 USENIX Annual Technical Conference (USENIX ATC 04). Boston, MA: USENIX Association; 2004. .

[16] Organization LK. Util-linux, version 2.36.2; 2020. Size: 3.48 MiB, Contains fdisk utility. Available from: https://cdn.kernel.org/pub/linux/utils/util-linux/v2.36/util-linux-2.36.2.tar.xz.

[17] Foundation FS. The GNU C Library (glibc), version 2.32; 2020. Size: 196.47 MiB, Referred to as Gibbc in the paper. Available from: https://ftp.gnu.org/gnu/glibc/glibc-2.32.tar.xz.

[18] Foundation FS. GNU Automake, version 1.16.3; 2020. Size: 8.69 MiB. Available from: https://ftp.gnu.org/gnu/automake/automake-1.16.3.tar.xz.

[19] Organization LK. Linux Kernel Source, version 5.10; 2021. Size: 824.5 MiB, Referred to as LKT in the paper. Available from: https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.tar.xz.

[20] Foundation FS. GNU Coreutils, version 8.32; 2020. Size: 9.85 MiB. Available from: https://ftp.gnu.org/gnu/coreutils/coreutils-8.32.tar.xz.

[21] Foundation TAS. Apache Cassandra Source Release, version 4.0-beta3; 2020. Size: 1,409 MiB. Available from: https://downloads.apache.org/cassandra/4.0-beta3/apache-cassandra-4.0-beta3-src.tar.gz.

[22] Community TS. Squeak Smalltalk System Image; 2021. Size: 45.10 MiB. Available from: http://files.squeak.org/trunk/Squeak6.0-2021-07-02-123456.zip.

[23] Authors TC. Chromium Source Code Snapshot; 2021. Approximate size: 14,627 MiB. Available from: https://chromium.googlesource.com/chromium/src.git/+archive/{commit-hash-or-date}.tar.gz.

[24] Foundation FS. Bash, version 5.1; 2020. Size: 116.44 MiB. Available from: https://ftp.gnu.org/gnu/bash/bash-5.1.tar.gz.