Title: CSA Archiver: A High-Performance Multi-Threaded File Compression System Author: Ahmed Elashry

---

Abstract: This paper presents the design and implementation of the CSA Archiver, a custom file compression system developed in Python with PyQt5 for GUI management. The system is designed to efficiently compress large file hierarchies, provide an integrated folder-aware explorer, and optimize throughput using multi-threaded compression. The core algorithm leverages a bespoke compression pipeline, maintaining an indexed archive for fast access. The focus is on system performance, maintainability, and modularity, while remaining accessible for high-level technical inspection.

1. Introduction Compression of digital files is essential for storage efficiency and transfer speed. While traditional archivers like ZIP, RAR, and 7z offer widespread use, they often rely on generic compression strategies that do not leverage the specific structure of file hierarchies. The CSA Archiver addresses this gap by implementing a compression pipeline that integrates multi-threading, fine-grained indexing, and a graphical explorer, allowing users to navigate archives like file systems without extraction.

2. System Architecture The CSA Archiver consists of three primary modules: a PyQt5-based GUI, a compression core, and an extraction engine. The GUI contains two main tabs: Compressor and Explorer. The compressor tab allows the user to select a source folder, choose output location, and launch multi-threaded compression. The explorer tab enables opening a CSA archive, displaying folder hierarchy, and extracting files.

The system uses Python's concurrent.futures.ThreadPoolExecutor to manage multiple worker threads. Each thread handles individual file compression, ensuring maximal CPU utilization while maintaining thread-safe writes to the output archive.

1. Compression Core All files are processed via compress_file_core, a Python function implementing generic compression strategies. Upon reading a file into memory, compress_file_core applies algorithmic transformations (e.g., LZ-based methods) to produce a compressed blob. For each file, metadata is recorded: original size, compressed size, method identifier, and additional file-specific attributes.

The archive format begins with a fixed 3-byte magic number (b'CSA'), followed by a 4-byte placeholder for index size. Compressed file blobs are written sequentially, and the JSON-formatted index is appended at the end of the file. This index allows rapid lookup of file positions and metadata without requiring decompression.

1. Multi-Threaded Compression The compression process creates a thread pool with N workers, where N is either user-specified or defaults to the minimum of 4 and the system's CPU core count. Each worker executes the compress_one function independently. Synchronization is handled by sequentially writing blobs to the archive, updating the current offset, and recording the metadata into a shared index dictionary. As_completed futures allow the main thread to update progress metrics and logs in real-time, providing responsive feedback via PyQt5 signals.

2. Explorer Module The explorer utilizes PyQt5's QTableWidget to represent archive contents in a folder-aware hierarchy. The JSON index maps relative file paths to metadata, allowing the GUI to display original and compressed sizes, compression ratios, and methods. This module allows

users to inspect archive contents and extract files individually or entirely, maintaining directory structures.

3. Archive Format The CSA archive is a sequential layout consisting of:

4. Header: 3-byte magic + 4-byte index size placeholder
5. File blobs: compressed data for each file
6. Index: JSON object mapping relative paths to blob positions and metadata

Advantages include sequential write performance, easy extraction, and self-contained metadata. The JSON index allows human-readable inspection and potential cross-platform access.

1. Extraction Module Extraction reads the JSON index, iterates over the file entries, seeks to the correct offset in the archive, and writes decompressed data to the destination folder. Multi-threading is optionally used to parallelize extraction for large datasets.

2. Performance Analysis The CSA Archiver demonstrates efficient CPU usage due to multi-threading. Sequential writes prevent fragmentation. Memory consumption is moderate, with one file loaded per worker. Compared to generic ZIP, CSA offers faster compression for large folder hierarchies and provides folder-aware exploration without extraction.

3. Advantages and Disadvantages Advantages:

4. Responsive, folder-aware GUI
5. Multi-threaded compression and extraction
6. Self-contained metadata index
7. Flexible architecture for additional compression methods

Disadvantages: - Custom archive format reduces compatibility with standard tools - JSON index adds minor overhead - Not optimized for maximum compression ratios like 7z or advanced LZMA variants

1. Implementation Details
2. GUI: PyQt5 with QTabWidget for compressor/explorer tabs
3. Threading: concurrent.futures.ThreadPoolExecutor
4. Compression Core: compress_file_core handling all compression, independent of RSF
5. Error Handling: Exceptions in individual files are logged; the archive continues

6. Indexing: JSON at end of archive; size stored in header for rapid access

7. Conclusion and Future Work The CSA Archiver provides a high-performance, multi-threaded compression system with an integrated folder-aware GUI. Future work includes supporting hybrid compression algorithms (LZMA/ZLIB), finer-grained progress reporting, context menu support in explorer, and potential cross-platform portability.

References 1. Python Documentation, concurrent.futures. https://docs.python.org/3/library/concurrent.futures.html 2. PyQt5 Documentation. https://www.riverbankcomputing.com/static/Docs/PyQt5/ 3. Zlib Compression Library. https://www.zlib.net/ 4. LZMA SDK. https://tukaani.org/xz/ 5. K. Sayood, Introduction to Data Compression, 5th Edition, 2017.