Documentation for Deribit Trading system

1. Benchmarking -

Benchmarked the following metrics -

- **Order Placement Latency:** Time taken from placing an order to confirmation.
- **Market Data Processing Latency:** Time taken to process incoming market data updates.
- **WebSocket Message Propagation Delay:** Time for messages to travel between the server and client.
- **End-to-End Trading Loop Latency:** Total time from market data input to order execution.

Methodology used -

1. Order Placement Latency

- **Function:** `BenchmarkPlaceOrder`
- **Process:**
  - Redirects standard input/output to simulate user input for placing an order.
  - Calls `trading_system.placeOrder()` in a loop to measure the average latency of order placement.
- **Purpose:** Evaluate the time it takes for the system to process and confirm an order.

2. Market Data Processing Latency

- **Function:** `BenchmarkMarketData`
- **Process:**
  - Simulates market data subscription using mocked input.
  - Measures the time taken to fetch and process market data for the subscribed channel.
  - Unsubscribes after the benchmark.
- **Purpose:** Identify delays in receiving and processing market data from the exchange.

3. WebSocket Message Propagation Delay

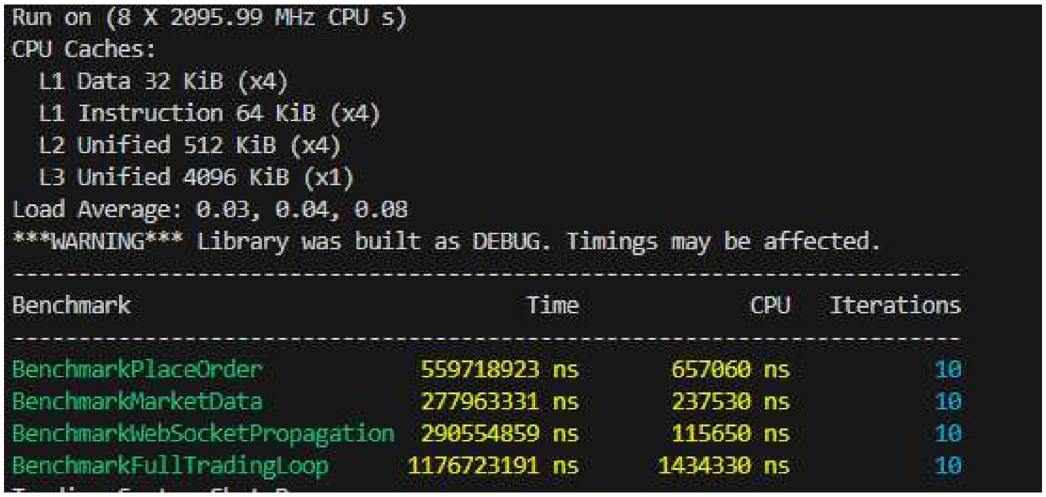
- **Function:** `BenchmarkWebSocketPropagation`
- **Process:**
  - Sends a test message (`public/test`) over WebSocket and measures the time to receive a response.
- **Purpose:** Quantify the latency introduced by WebSocket communication, including transmission and server-side processing.

4. End-to-End Trading Loop Latency

- **Function:** `BenchmarkFullTradingLoop`
- **Process:**
  - Simulates a complete trading loop:
    1. Places a buy order.
    2. Immediately places a sell order in the same iteration.
  - Redirects I/O streams to simulate user inputs for each action.
- **Purpose:** Measure the overall latency from decision-making to order placement and execution.

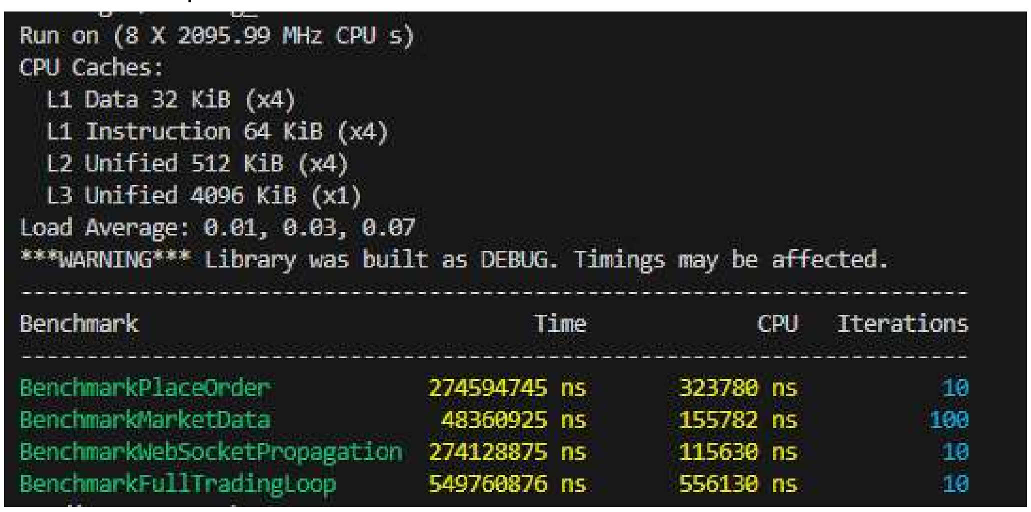
Results of Benchmarking:

Before Optimization (memory optimization)-



```
Run on (8 X 2095.99 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 64 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 4096 KiB (x1)
Load Average: 0.03, 0.04, 0.08
***WARNING*** Library was built as DEBUG. Timings may be affected.
-------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations
-------------------------------------------------------------------------
BenchmarkPlaceOrder              559718923 ns       657060 ns           10
BenchmarkMarketData              277963331 ns       237530 ns           10
BenchmarkWebSocketPropagation    290554859 ns       115650 ns           10
BenchmarkFullTradingLoop        1176723191 ns      1434330 ns           10
```

After Optimization-



```
Run on (8 X 2095.99 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x4)
  L1 Instruction 64 KiB (x4)
  L2 Unified 512 KiB (x4)
  L3 Unified 4096 KiB (x1)
Load Average: 0.01, 0.03, 0.07
***WARNING*** Library was built as DEBUG. Timings may be affected.
-------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations
-------------------------------------------------------------------------
BenchmarkPlaceOrder              274594745 ns       323780 ns           10
BenchmarkMarketData               48360925 ns       155782 ns          100
BenchmarkWebSocketPropagation    274128875 ns       115630 ns           10
BenchmarkFullTradingLoop         549760876 ns       556130 ns           10
```

2. Optimization Study -

These were the practices implemented for optimising the overall latency of the system and optimise the memory used

1. **Centralized Fixed-Length Data Structure for Subscriptions**: Utilizing a fixed-length array or data structure allows for constant-time access to subscription data, improving retrieval speed. This approach ensures that each subscription can be accessed directly without the need for searching or additional computations.

2. **Use of `const` in Functions**: Declaring member functions as `const` indicates that these functions do not modify the object's state. This practice enhances code clarity and allows such functions to be invoked on `const` instances of the class. It also enables the compiler to perform optimizations and ensures that the function does not alter the object's data.

3. **Asynchronous Network Operations with Boost.Asio**: Employing Boost.Asio facilitates asynchronous network operations, allowing the program to handle multiple tasks concurrently without blocking. This is particularly beneficial for I/O-bound operations, as it enables the application to remain responsive and efficient.

4. **Dedicated Threads for WebSocket Connections**: Assigning separate threads to manage WebSocket connections ensures that each connection operates independently, preventing blocking and improving scalability. This multithreading approach allows the application to handle multiple WebSocket connections simultaneously, enhancing performance and responsiveness.

5. **Separate Thread for Processing Market Data**: Isolating market data processing in its own thread allows for concurrent execution, reducing latency and improving throughput. This design pattern is effective in scenarios where data processing is intensive and can benefit from parallel execution.

6. **WebSockets for Persistent Connections**: Utilizing WebSockets establishes a persistent connection between the client and server, eliminating the need for repeated TCP handshakes. This persistent connection reduces latency and overhead, allowing for real-time data exchange and efficient communication.

By integrating these practices, the code achieves enhanced performance, scalability, and maintainability, making it well-suited for applications requiring efficient data handling and real-time communication