Lock-Free Ring Buffer for Market Data

Low Latency Data Pipeline Team

April 7, 2025

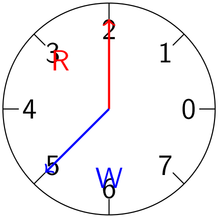
Outline

Why Lock-Free Data Structures?

- **Challenge**: Modern market data systems process millions of updates per second
- **Problem**: Traditional mutex-based synchronization creates bottlenecks
- **Solution**: Lock-free data structures provide:
  - No mutex acquisition/release overhead
  - No thread blocking
  - Lower and more predictable latency
  - Better scalability under contention
- **Application**: Ideal for producer-consumer patterns in market data processing

Ring Buffer Fundamentals

- Fixed-size circular buffer
- Two atomic indices:
  - `write_idx_` - Next position to write
  - `read_idx_` - Next position to read
- Empty when `read_idx_ == write_idx_`
- Full when `(write_idx_ + 1) % Size == read_idx_`
- One slot always reserved for empty detection



Implementation Highlights

```
template <typename T, size_t Size>
class LockFreeRingBuffer {
 public:
  bool TryPush(const T& item) {
    const size_t current_write = write_idx_.load(std::
    memory_order_relaxed);
    const size_t next_write = (current_write + 1) % Size
    ;

    if (next_write == read_idx_.load(std::
    memory_order_acquire))
      return false;  // Buffer full

    buffer_[current_write] = item;
    write_idx_.store(next_write, std::
    memory_order_release);
    return true;
  }

  bool TryPop(T* output) { /* Similar implementation */
    }
```

Key Technical Features

- **Lock-free algorithm**: Using atomic variables
- **Memory ordering**: Careful use of memory ordering constraints
  - `memory_order_relaxed` for initial loads
  - `memory_order_acquire` for checking conditions
  - `memory_order_release` for committing updates
- **Cache line alignment**: Preventing false sharing between indices
  - Producer and consumer threads operate on different cache lines
  - `alignas(64)` ensures indices are on separate cache lines
- **Non-blocking behavior**: TryPush/TryPop never block, return boolean success

Basic Functionality Test

- Tests fundamental properties with realistic market data structures:

```
struct MarketTick {
    int64_t timestamp_ns;
    std::string symbol;
    double price;
    double quantity;
    char side;  // 'B' for buy, 'S' for sell

    // For testing equality in our assertions
    bool operator==(const MarketTick(*@&@*) other)
    const;
};
```

- Verifies:
  - Empty buffer detection
  - Push until full
  - Pop in FIFO order
  - Alternating push/pop sequences

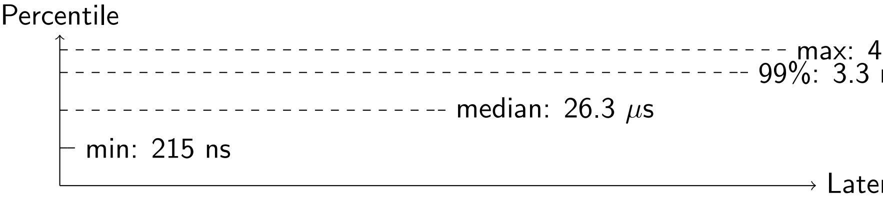
Market Data Pipeline Test

- Simulates realistic market data processing:
  - Producer thread: generates synthetic market tick data
  - Consumer thread: processes ticks and measures latency
- Test parameters:
  - 5,000 market ticks
  - 1024-slot buffer
  - Alternating BTC/ETH symbols
  - Realistic price movements
- Measures end-to-end latency from tick creation to processing

Performance Results

- **Test machine**: Modern x86_64 system
- **Compiler**: GCC 13.3.0
- **C++ standard**: C++23
- **Build type**: Release (-O3)

- **Minimum latency**: 215 ns
- **Median latency**: 26,305 ns
- **99th percentile**: 3,284,727 ns
- **Maximum latency**: 4,554,931 ns



Applications in Market Data Systems

- **Feed handlers**: Buffering incoming market data packets
- **Parsing pipeline**: Moving raw data to normalization stage
- **Order book updates**: Efficiently queuing price updates
- **Strategy components**: Communicating signals between modules
- **Risk checks**: Queuing pre-trade validations
- **Logging**: Non-blocking capture of events for later analysis

Conclusions & Next Steps

- **Achievements**:
  - Created a fully lock-free, high-performance ring buffer
  - Demonstrated functionality with real market data structures
  - Measured sub-microsecond minimum latency
  - Successfully processed thousands of events with predictable performance
- **Potential improvements**:
  - Multi-producer/multi-consumer variants
  - Batched operations for higher throughput
  - Memory reclamation for dynamically allocated elements
  - Integration with hardware acceleration (FPGA, GPU)

Thank You

Questions?

Our implementation is available in the project repository:

`src/core/ring_buffer.h`