Lock-Free Trade Engine Acceleration Through High-Efficiency Memory Pooling

Jhonathan Herrera

Thur Apr 24

1 Abstract

This project explores how multi-threading techniques and optimized memory management can significantly enhance the execution speed and overall performance of trade engines. One focus is on how pre-allocating memory via memory pools can reduce the overhead of frequent dynamic allocations and minimize CPU calls during runtime. Trade engines face the constant challenge of maintaining low-latency execution while maximizing throughput and minimizing CPU usage, which can cause delays and reduce system efficiency. By incorporating lock-free multi-threading and smarter memory handling strategies, this project aims to reduce resource consumption and boost execution performance.

The implementation leverages C++ standard libraries, such as <thread> and <atomic>, to enable concurrency, and integrates lock-free data structures to avoid traditional mutex bottlenecks. Additionally, tools like Valgrind, Leaks, and Thread Sanitizer will be used to identify performance bottlenecks and memory leaks during development.

The hypothesis is that these techniques, particularly avoiding std::mutex locks and using smart pointer-based memory pooling, will lead to noticeable performance gains, potentially saving fractions of a second per operation, which is highly impactful in trading systems. These optimizations are especially relevant for firms like AlgoGators, where even minor improvements in execution speed can result in substantial financial advantages.

2 Introduction

The performance of trade engines is critical for AlgoGators, where even small delays in execution can lead to significant financial consequences. This project explores how multi-threading and optimized memory management can improve the speed and efficiency of trade engine operations. By using techniques like RAII (Resource Acquisition Is Initialization), lock-free data structures, and memory pooling, the system aims to minimize latency, reduce CPU

usage, and eliminate bottlenecks caused by traditional locking mechanisms. In particular, when dealing with a predictable or fixed number of tasks, pre-allocating memory upfront instead of relying on repeated heap allocations can lead to faster execution and better resource control.

The hypothesis is that applying these optimization techniques will result in a noticeable improvement in execution time and overall performance. Early testing with tools such as Valgrind and ThreadSanitizer has already revealed memory inefficiencies and race conditions, which these techniques help address. Custom implementations like lock-free queues and optimized sorting algorithms also show potential for reducing overhead and improving throughput. Ultimately, this research supports AlgoGators' goal of achieving low-latency, high-efficiency execution in competitive trading environments, where even a few milliseconds can make a difference

3 Methodology

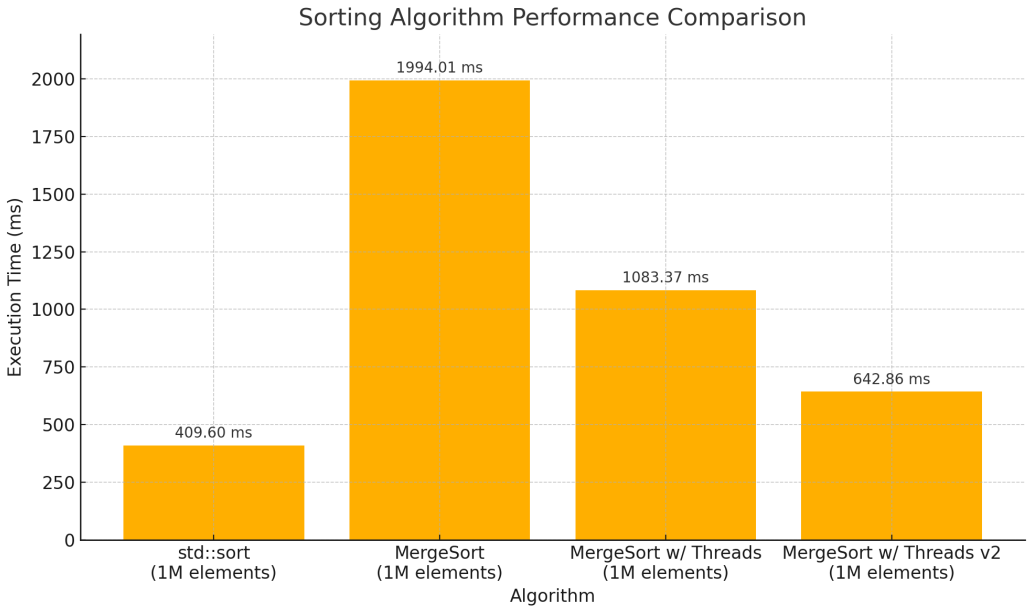
Due to technical limitations with the actual Trade-ngn system, direct testing on the live engine was not feasible. Instead, multiple Docker containers were made based on various Algo Trade-ngn scripts, which served as mock environments to simulate data input streams. These simulations allowed for realistic load testing, approximating the behavior of the production system under high input volumes.

Inside the Docker environment, diagnostic tools were used such as Valgrind and Thread Sanitizer, despite macOS compatibility challenges. These tools were instrumental in identifying memory leaks, observing thread behavior, and pinpointing potential synchronization inefficiencies. The analysis focused on performance benchmarking, utilizing high-resolution timers (std::chrono in C++) to measure execution time, track memory allocations, and detect race conditions. The performance of multi-threaded implementations were compared with their single-threaded counterparts, using terminal profiling outputs to assess system-level metrics. This

analytical approach helped quantify the benefits of multi-threading and lock-free structures, providing valuable insights into potential optimizations for the Trade-ngn engine.

4 Multi-Threading Preparation

As a preparatory step, a few evaluations were set to see how multi-threading could improve the performance of an execution function by implementing multiple versions of a merge sort algorithm, each employing different threading strategies. This self-assigned task served as foundational practice, given my limited prior experience with concurrency. The objective was to reduce latency as much as possible. The result was a 68% decrease in execution time compared to the single-threaded baseline. Although the performance still does not match the efficiency of `std::sort()`, this experiment demonstrated the significant potential for optimization within the engine. Its believed that further refinements, especially with more advanced thread management and memory reuse techniques, could yield even greater improvements in performance.

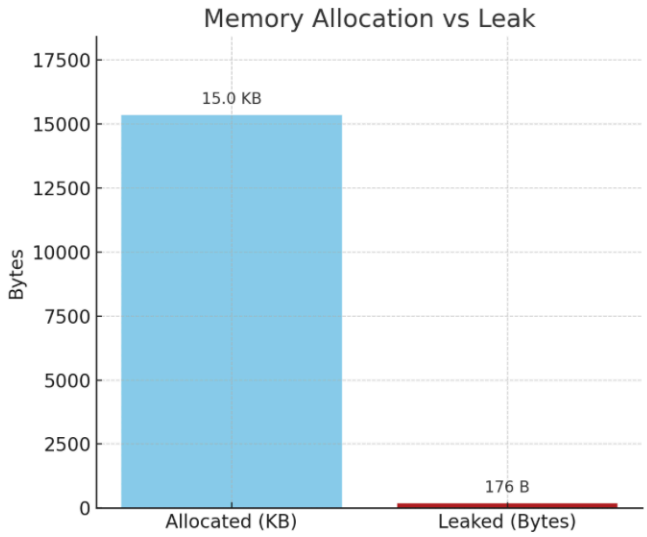


Comparison of four sorting algorithms (standard and multithreaded implementations), measured in milliseconds (ms)

5 Memory Safety & Optimization

5.1 RAII and Smart Pointers Overview

To better understand the importance of memory safety, tests were conducted by intentionally creating memory leaks through raw pointer usage and bypassing RAII (Resource Acquisition Is Initialization). Tools like Valgrind and Leaks successfully flagged these leaks, allowing me to observe how unmanaged heap allocations can negatively impact performance and system stability. This testing underscored the importance of using RAII principles and smart pointers to automatically manage resources, especially in high-performance systems like a trade engine where reliability and speed are critical



5.2 Why Smart Pointers Matter in a Trade Engine

In algorithmic trading systems, execution speed and memory efficiency directly affect latency and reliability. Trade engines often handle large volumes of data in real time, with dynamic memory allocation for orders, signals, and market updates. Smart pointers reduce the risk of memory leaks, dangling pointers, and double deletions, which are especially dangerous in multithreaded or long-running applications. Additionally, managing memory through RAII

means that resources are acquired and released in a deterministic way, helping avoid crashes or excessive memory usage during high-frequency operations. This leads to more robust, readable, and maintainable code, which is vital in performance-critical financial systems.

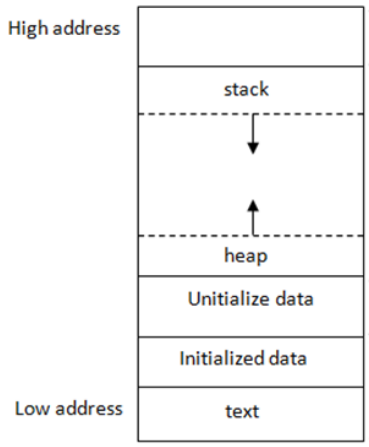


Figure 1. C/C++ Program Memory Layout showing the relationship between the stack and heap (University of Illinois at Urbana-Champaign, 2022)

Smart Pointer	Ownership	Copy/Move Behavior	Memory Management
Std::unique_ptr	Exclusive	Cannot be copied, only moved	Automatically deletes object when out of scope
Std::shared_ptr	Shared	Can be copied and moved	Deallocated when the last shared_ptr is destroyed

6. Smarter Pointer Limitations and the Case for Memory Pools

Although smart pointers are excellent for managing memory safely and automatically handling deallocation and preventing leaks, they can become inefficient in high-load systems. In a large engine that processes millions of transactions or elements, repeatedly creating shared_ptr or unique_ptr objects leads to constant dynamic memory allocations. This results in significant

overhead, increasing CPU workload and potentially causing performance bottlenecks such as cache thrashing or memory fragmentation.

To address this, a more efficient alternative is to pre-allocate a fixed block of memory when you already know the approximate data size you'll be handling. By using a memory pool, you can allocate and deallocate memory from this pool and recycle bytes, drastically reducing the number of heap allocations. This approach minimizes system calls, improves cache locality, and delivers more consistent execution times, making it ideal for latency-sensitive applications like trading engines.

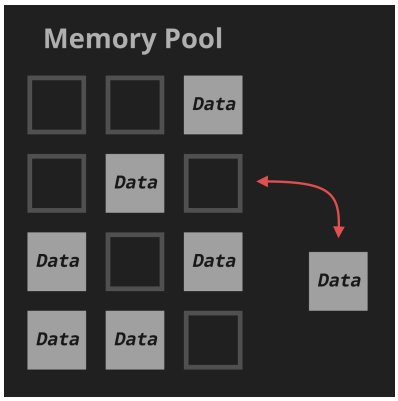
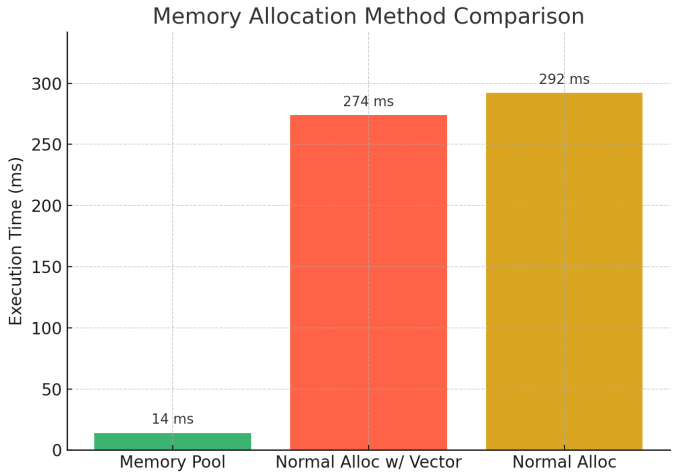


Figure 2. Memory Pool (Public Domain via CC0 1.0)

In a test script called portfolio_minimal_test.cpp, a mock DummyStrategy was created that runs across 4 threads, each executing 10 rounds of processing with 250,000 allocations per round. Altogether, this resulted in 10 million total allocations across all threads. To optimize memory usage and reduce overhead, a custom memory pool that pre-allocates 250,000 bytes per round was implemented. This allowed the engine to allocate and deallocate memory efficiently by recycling memory instead of repeatedly requesting new allocations from the heap. This approach proved to be much more efficient than using smart pointers like std::shared_ptr, which can introduce significant CPU overhead when handling large volumes of data and frequent allocations.

When timing the execution speed, we can see that the memory pool completed in just 14 ms, compared to 274 ms for vector-based allocation and 292 ms for standard new/delete allocation. This means the memory pool was approximately 20.8X faster, or about 95.2% more efficient, than the slowest method. This improvement is largely due to avoiding repeated heap allocations and reducing CPU overhead. By recycling memory instead of making frequent system calls, the memory pool significantly improves performance. While memory pooling is most beneficial in systems handling large volumes of data, this result demonstrates that, when implemented correctly, it can be a powerful and efficient alternative to traditional allocation strategies.



7 Synchronization Strategies

7.1 Mutex Based Threading

In C++, you can execute different tasks in parallel using threads. A good way to visualize this is to imagine two chefs in a kitchen working on the same dish. By splitting up the work, the dish might be prepared in half the time. However, if the chefs don't coordinate properly, they might bump into each other or redo the same step. This is exactly what can happen in multithreaded

programs, when multiple threads try to access or modify the same data at once, race conditions and data corruption can occur.

To solve this, C++ provides mutexes and std::lock_guard. A mutex (mutual exclusion) ensures that only one thread at a time can access a critical section of code or data. When a thread enters a mutex-protected section, all other threads must wait until it finishes and releases the lock. This prevents interference but comes with a tradeoff: threads might sit idle waiting for access, which can become a bottleneck, especially in high-performance systems like trade engines that demand ultra-low latency.

This type of thread waiting can be compared to Serializable isolation in databases, where transactions are strictly ordered for consistency but at the cost of performance. Similarly, in multithreaded code, mutexes enforce order but introduce blocking, which can slow things down.

7.2 Lock-Free with Atomics

Instead of using a mutex, we can use std::atomic, which allows threads to read and write to variables safely without locking. Atomics are low-level operations provided by the CPU that ensure memory consistency without requiring a full lock. They're ideal for small, simple data types like integers, flags, and pointers.

With atomics, you can write lock-free (or "free-lock") code, which means threads don't have to wait for each other. If one thread is working on data, another thread can keep moving forward, retrying or working on other parts of the task. This dramatically improves throughput in systems that can't afford to pause.

Lock-free programming is typically built around CPU-level atomic instructions, such as compare-and-swap (CAS) or C++'s compare_exchange_weak and compare_exchange_strong. This works by comparing a variable to an expected value and, if they match, updating it to a new value in one atomic step. If another thread has modified the variable in the meantime, the operation fails and the thread simply retries , no waiting, no blocking. This retry loop continues

until the update succeeds, allowing the system to make progress without ever pausing a thread. By avoiding locks entirely, lock-free algorithms reduce contention, eliminate deadlocks, and improve performance in high-concurrency environments.

#threads	xchg			xadd			cmpxchg		
	PC_A	PC_B	PC_C	PC_A	PC_B	PC_C	PC_A	PC_B	PC_C
1	17.6	16.4	18.8	19.3	19.1	18.8	23.3	21.1	18.8
2	18.6	16.4	18.8	91.0	134.8	60.5	96.5	284.4	60.2
4	-	16.4	19.5	-	572.7	124.8	-	572.9	126.4
8	-	-	37.8	-	-	255.3	-	-	267.1
16	-	-	37.8	-	-	679.6	-	-	692.2

```
1 retry:
2   while (lock) backoff()
3   locked = xchg(&lock,1)
4   if (locked) goto retry
5 val=val+1
6 lock=0
```

```
1 retry:
2   c=val;
3   c1=c+1
4   ret=cmpxchg(&val, c, c1)
5   if (ret!=cur)
6     backoff(b)
7     goto retry
```

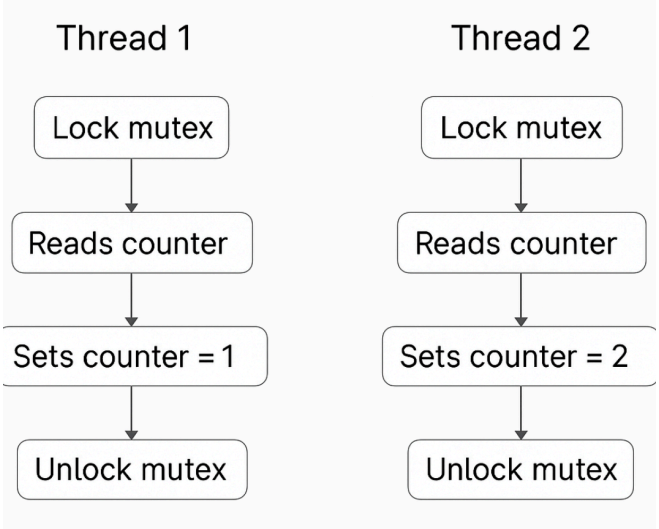
xchg, xadd and cmpxchg instruction execution time in processor cycles

To implement this, a signal minimal benchmark using two versions of a signal engine were ran. Both engines processed 3,200,000 signals generated in parallel by 32 threads, each sending 100,000 signals. One version used a traditional std::mutex for synchronization, while the other used an atomic-based, lock-free approach. The mutex-based engine completed in 6450 ms, while the free-lock engine finished in 5498 ms, resulting in a 14.7% decrease in execution time.

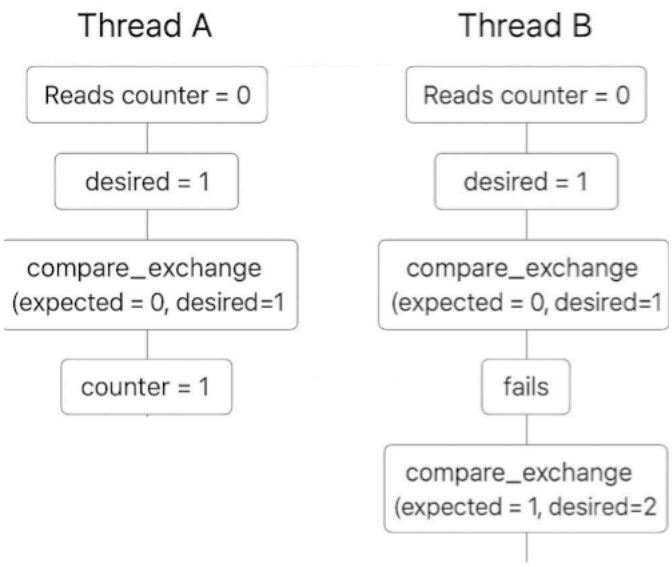
This improvement is due to the lock-free engine allowing threads to update shared memory without blocking, which avoids costly context switching and reduces contention. In the mutex-based version, every thread must wait its turn to acquire the lock before appending to the signal vector, which causes idle time and serialization under heavy load. In contrast, the lock-free version leverages std::atomic<std::shared_ptr> to handle shared ownership without locking, enabling multiple threads to make progress simultaneously.

These results demonstrate the clear performance advantage of lock-free techniques in high-throughput systems like trade engines, where minimizing latency and maximizing parallelism are critical. Of course, while lock-free programming introduces efficiency, it also requires careful design to avoid issues like race conditions, memory reclamation challenges, and retry loops. When analyzing the graph below, we observe that execution time does not consistently improve with an increasing number of threads. This highlights that more threads do not always equate to better performance. When performing with too many threads, it can cause cache invalidation and traffic which can slow down performance, additionally if you have more threads than CPU core. Instead, it's crucial to determine the optimal number of threads tailored to the specific needs and behavior of your application. Typically aim for the amount of threads thats less than the amount go CPU core the masOS/Linux has. But when applied correctly, as shown in this test, lock-free structures can significantly reduce bottlenecks and boost scalability.

Mutex Locked Based Counter



Atomic Free-Lock Based Counter



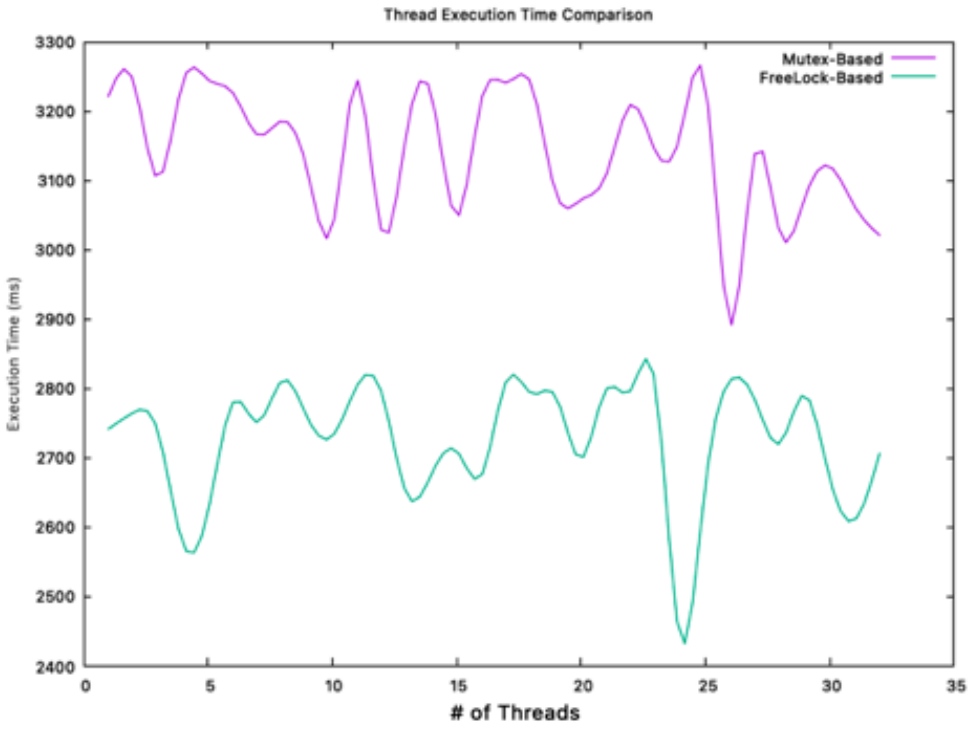

```
// Mutex-based version
mutable std::mutex mtx_;

size_t num_signals() const {
    std::lock_guard<std::mutex> lock(mtx_);
    return signals_.size();
}

// Lock-free atomic version
std::atomic<size_t> index_;

size_t num_signals() const {
    return index_.load(std::memory_order_relaxed);
}
```

Same code function, The top is using mutex and the bottom uses atomic and free-lock



8 Holes & Ugly Risks

While the implementation of a lock-free engine with memory pooling offers substantial performance benefits, it also introduces potential vulnerabilities and design tradeoffs that must be acknowledged. These "holes" refer to gaps in logic or safeguards, such as the absence of a fallback mechanism when the memory pool is exhausted, which could result in crashes or undefined behavior during peak loads. Meanwhile, "ugly risks" encompass less visible but equally important concerns, such as inefficient smart pointer usage under high-frequency allocation patterns, or the lack of runtime observability which can mask performance degradation. Though these issues may not immediately impact functionality, they can lead to unpredictable behavior, technical debt, or bottlenecks in production environments. Identifying and addressing these risks early is critical to building a system that is not only fast but also stable and maintainable under real-world conditions.

9 Memory Reclaiming

9.1 ABA Issues

A trading engine that combines memory pooling with lock-free data structures can deliver exceptional performance by reducing heap allocations and minimizing contention across threads. However, even when using atomic operations and compare-and-swap (CAS) to protect shared data, critical issues can still occur when memory is recycled too early, issues that CAS alone cannot prevent.

Consider a lock-free stack with Node A and two threads: Thread A and Thread B.

1. Thread A pops Node A and returns it to the memory pool.
2. Thread B, which also previously loaded Node A, proceeds with operations under the false assumption that the node's content is still valid.
3. Meanwhile, Thread A reuses the memory from the pool and writes new data into Node A.

At this point, Thread B is accessing recycled memory thinking it's still working with the original Node A, when in reality, that memory has been repurposed with new content.

This leads to undefined behavior, including data corruption, security vulnerabilities, or crashes.

This situation defines the classic ABA problem: a memory address goes from A → B → A, giving the illusion that it hasn't changed when, in fact, the contents have.

Since CAS only compares addresses, it cannot detect if a node was removed, recycled, and reinserted at the same location — making it insufficient on its own.

9.2 Memory Reclamation Techniques

The process of recycling memory in concurrent systems is known as memory reclamation (MR). In lock-free and low-latency applications like trading engines, effective MR is essential for maintaining performance while avoiding memory leaks or unsafe reuse. Several well-established MR algorithms exist to address issues like the ABA problem and premature memory reuse. These include techniques such as hazard pointers, epoch-based reclamation, and reference counting , all designed to ensure that memory is only reclaimed when it is guaranteed to be

unused by any thread. Implementing MR correctly is not only possible but critical for ensuring the safety and correctness of high-performance multithreaded systems.

9.2.1 Hazard Pointers

To solve this, one approach is to use hazard pointers, a safe memory reclamation technique specifically designed for lock-free data structures.

Here's how they work:

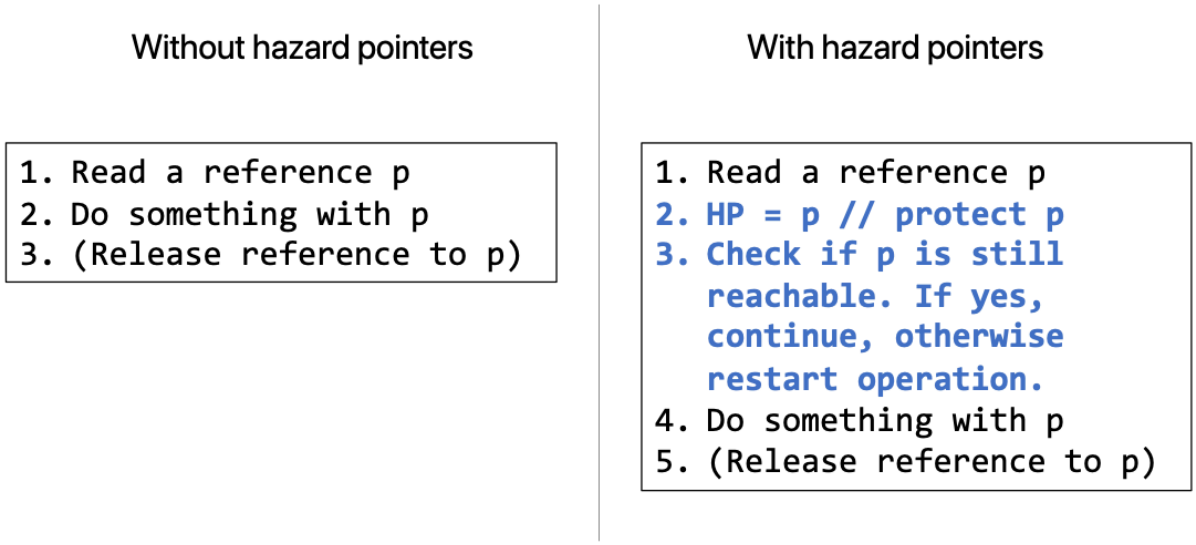
1. Before accessing a shared node, a thread sets a hazard pointer to that node's address. This marks the memory as in use, signaling to other threads that it must not be freed or reused yet.
2. When another thread attempts to remove and recycle a node, it checks all hazard pointers across threads:

- If the node is still referenced, it is not reclaimed immediately.
- Instead, the thread places it into a retired list (sometimes called a "limbo list"), which is a temporary storage for nodes waiting to be safely recycled.

3. Periodically, each thread scans all hazard pointers:

- If a node in the retirement list is no longer referenced by any hazard pointer, pit is finally safe to delete or return to the memory pool.

Hazard pointers provide fine-grained, thread-level protection, preventing memory from being reclaimed while it's still in use, and offering a reliable defense against ABA-related issues in lock-free systems.



Instruction execution time for xchg, xadd, and cmpxchg in processor cycles.

9.2.2 Lock-Free Reference Counting

Reference counting is similar to how std::shared_ptr works , it keeps track of how many references are currently held to a specific object or memory block. When the reference count drops to zero, the memory can be safely reclaimed. Just like with shared_ptr, the memory is automatically deleted when there are no remaining owners.

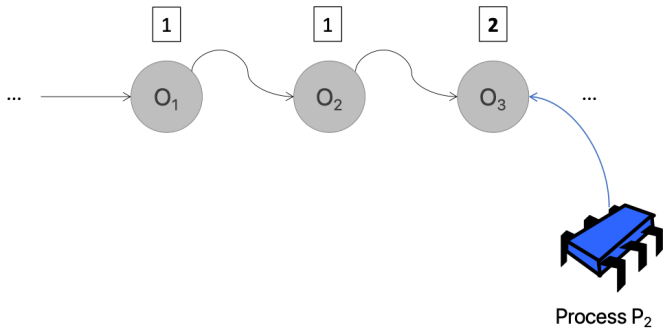
In a multithreaded system, every time a thread starts using a piece of memory or an object, it atomically increments the reference count. When the thread is finished, it decrements the count. Once the count reaches zero, the object is recycled or returned to the memory pool safely.

Within a memory pool, it's crucial to ensure that no thread allocates memory that is still in use. Lock-free reference counting helps enforce this by only returning memory to the pool once the reference count is zero, meaning no thread is currently using it. This prevents data corruption, all without needing locks or mutexes, making it ideal for low-latency systems like trading engines.

It’s a relatively easy-to-understand and implement concept, but it comes with some trade-offs:

1. Performance can suffer due to the need update the references counter on every access (both reads and writes)

2. Each increment and decrement requires atomic CPU instructions, which can add overhead, especially in large systems where minimizing latency is crucial



Nodes with reference Counters Reflecting Concurrent Reads and Writes by Other Processes

9.2.3 Epoch-Based Reclamation (EBR)

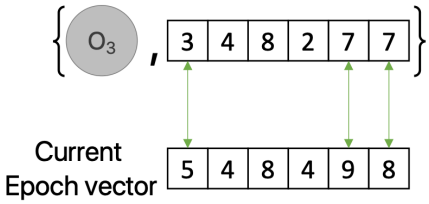
Epoch-Based Reclamation (EBR) is a memory management strategy designed for concurrent systems, where memory is only reclaimed after it is guaranteed that no threads are still accessing it. EBR works by tracking the progress of all threads interacting with shared memory, ensuring safe and efficient reclamation without requiring per-object reference tracking. Each thread maintains a local epoch value, typically represented as an integer. Upon entering a critical section, the thread records the current global epoch, marking its activity. An odd epoch value indicates the thread is currently inside a critical section, while an even value signifies it has exited.

When memory is removed from a shared structure, it is not immediately freed. Instead, it is tagged with the current global epoch and placed into a retirement list (also known as a "limbo list"). Periodically, the system scans these lists and checks the epoch values of all threads. If all threads have either exited or advanced beyond the epoch in which the memory was retired, the memory can then be safely reclaimed and returned to the memory pool.

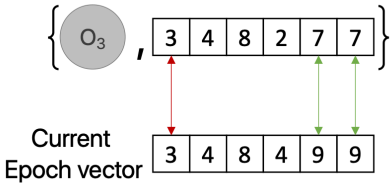
EBR offers efficient performance with relatively low overhead, as it avoids costly synchronization or fine-grained reference counting. However, it is not entirely lock-free: if a thread becomes delayed or fails to exit its critical section, it may block memory reclamation for all other threads, leading to potential memory retention and degraded performance. Nonetheless, EBR remains a practical and scalable solution for memory reclamation in high-performance, multithreaded environments.

Scan:

- cur_vec = current epoch vector
- For each node *n* in the limbo list:
  - node_vec = n's epoch vector
  - For each process i:
    - if node_vec[i] is odd
      - if node_vec[i] >= cur_vec[i]
        - Continue to next node
  - Free node



OK to reclaim!



Not OK to reclaim!

10 Fragmentation Threat

Although a memory pool is beneficial, it can present issues when using fixed-size blocks to allocate memory. If the requested memory size doesn't match the block size, it can lead to fragmentation and inefficient memory usage. However, there are specific techniques designed to overcome these limitations and improve memory allocation efficiency.

10.1 Slab Allocator

A slab allocator is a memory management strategy that pre-allocates large memory chunks (called slabs), where each chunk is divided into fixed-size blocks that match the size of a specific object type (e.g., Order, Signal). Unlike general-purpose allocators that may allocate memory of arbitrary sizes and cause fragmentation, a slab allocator is optimized for reuse of one consistent size — making it efficient and cache-friendly.

If a memory pool allocates blocks that are larger or smaller than the objects being stored, this can lead to fragmentation. A slab allocator avoids this by matching the block size to the object's actual size. In C-like syntax, slab allocation feels similar to using malloc(sizeof(T) * N), but it's more structured: it tracks which slots are free, reuses them, and often avoids calling malloc repeatedly. It’s like building a custom allocator that gives you N pre-allocated objects of size sizeof(T) and lets you recycle them efficiently.

10.2 Segregated Freelist

A memory pool is an effective way to reduce CPU overhead when handling frequent memory allocation requests. Instead of repeatedly calling malloc or new, which can be computationally expensive, the memory pool pre-allocates a large block of memory and distributes it in fixed-size blocks.

However, a limitation of this approach is that every allocation returns a block of the same size, regardless of how much memory the object actually needs. This can lead to several issues like

wasted memory or internal fragmentation. To address this limitation, a technique called the Segregated Free List (SFL) can be used. An SFL enhances memory pooling by dividing the pool into multiple bins, each managing blocks of a specific size (e.g., 32B, 64B, 128B, etc.).

When an object is allocated, the requested size is rounded up to the nearest supported block size, and memory is pulled from the corresponding bin. For example, if an Order object requires 30 bytes, the allocator would provide a 32-byte block from the 32B bin. Likewise, a 100-byte request would be served by a 128B bin. This strategy preserves the performance benefits of a memory pool while reducing memory waste and improving allocation efficiency for variable-sized objects.

11 Fallback Mechanism

When using memory pooling techniques, there may be situations where the pool becomes exhausted, meaning all pre-allocated memory blocks have been used. If no backup strategy is in place, this can result in system crashes, undefined behavior, or the failure to handle critical tasks, especially in latency-sensitive systems like trade engines. A common and effective solution to this problem is to implement a fallback mechanism. When the pool is depleted, the fallback mechanism dynamically allocates memory using standard heap functions such as malloc or new to fulfill the request. Although these allocations are slower than those from a pool, they ensure that the system continues to operate rather than fail abruptly. While fallback mechanisms introduce additional complexity and may increase allocation time, they are crucial for maintaining system stability, especially during unpredictable spikes in activity.

12 Observability Gap and Logging

When working with large memory pools, certain forms of misbehavior can arise, especially as the pool nears exhaustion. Issues such as pool depletion, segmentation faults, or excessive reliance on fallback allocations may not immediately crash the system, but they can gradually erode performance or lead to instability. The critical problem is that these symptoms often go unnoticed until the system begins to behave erratically or fails entirely during execution.

To mitigate this risk, it's essential to implement runtime monitoring that logs and analyzes pool-related behavior. This includes tracking the current pool usage, the frequency of fallback allocations, and the latency of memory requests. By identifying these trends early—before they manifest as system faults, you can dramatically reduce the risk of crashes or missed executions in a trading engine, ensuring the system remains stable under stress and responsive in real-time conditions.

13 Discussion

So far, I've gained a deeper understanding of multi-threading, the RAII (Resource Acquisition Is Initialization)principle, lock-free programming, and various memory allocation techniques, along with their inherent complexities. One of the most challenging aspects of this research has been debugging and structuring a program using these tools. Since multi-threading doesn't offer a visual representation of how tasks are executed, it's often difficult to identify why certain issues occur or how threads interact. This lack of visibility makes diagnosing race conditions or logic errors particularly time-consuming.

Another significant challenge was using tools like Valgrind and Thread Sanitizer. While these tools are powerful for detecting memory leaks, data races, and synchronization issues, interpreting their output and applying effective fixes often isn't straightforward, especially

without a strong background in operating systems or low-level systems programming. Since I haven't taken courses in these areas yet, I recognize there's still a lot I don't fully grasp. However, I'm looking forward to diving deeper through future coursework and continued research to build a more complete mental model of what's happening under the hood.

Through experimentation, I also learned about different memory allocation strategies, such as manual heap management, smart pointers, and memory pools. These techniques directly impact performance, especially in systems handling millions of objects or requiring strict memory efficiency. Implementing lock-free data structures has also been difficult to wrap my head around. Understanding the atomic operations and memory ordering requirements takes time and practice. Given these challenges, I believe a future AlgoGators project could benefit from a real-time UI that visualizes memory leaks, thread states, and race conditions. Rather than parsing long terminal logs, developers could use a visual tool, possibly integrated into the AlgoLens service, to more quickly detect, debug, and resolve issues in the trade engine. This would significantly improve the developer experience and reduce time spent on low-level debugging.

Despite these challenges, the early results of my experiments support the hypothesis that multi-threading and optimized memory management can significantly improve the performance of trade engines. The development of visual debugging tools and enhanced memory monitoring will be critical in overcoming current limitations and achieving greater efficiency, reduced latency, and lower resource usage in future AlgoGators systems.

12 Conclusion

Tools like Sanitizer and Valgrind have been extremely helpful in identifying memory leaks and related issues during allocation testing. They provide precise insight into where leaks occur, how large they are (in bytes), how many exist, and the overall memory usage across files and threads. Moving forward, I plan to apply these tools to the actual engine once it's integrated, tracking the total number of leaks prevented and measuring the number of bytes and kilobytes saved through

optimized memory usage. The ultimate goal is to reduce memory leaks to zero without compromising CPU efficiency or performance.

Understanding these concepts in isolation is manageable, but applying them effectively within a system requires real experience and a deeper understanding of how everything works together. Optimization techniques like lock-free programming, smart pointers, and memory pools can be powerful tools, but only when used correctly. If misapplied, they can introduce complexity and new problems instead of solving them.

Even though I've spent a lot of time researching memory management and low-level system behavior, I haven't formally taken an Operating Systems or Computer Architecture course yet. Everything I've done so far has been self-taught. I plan to take an OS course next semester, and I'm hoping it will give me a stronger foundation to better understand what's happening "under the hood" and allow me to push my research further into more advanced and impactful topics.

Appendix

- .load() -> Atomically reads the current value
- .store(value) -> Atomically writes a new value
- .exchange(value) -> Atomically replaces value and returns the old one
- .fetch_add(x) -> Adds x to the atomic and returns the old value
- .fetch_sub(x) -> Subtracts x from the atomic and returns the old value
- .compare_exchange_weak(expected, desired) -> Tries to change the value to desired only if current value == expected.
- .compare_exchange_strong(expected, desired) -> same as exchange weak without the failing spuriously aspect to it.
- std::memory_order_relaxed -> No ordering guarantees. Fastest. Use when only atomicity matters
- std::memory_order_consume -> (Rarely Used) Data dependency ordering, often treated as acquire by compliers
- std::memory_order_acquire -> Prevent earlier loads from being moved after this read. Common for readers
- std::memory_order_release -> Prevent later stores from being moved before this write. Common for writers.
- std::memory_order_acq_rel -> Combines acquire + release. Use with compare_exchange often
- std::memory_order_seq_cst-> (Default) Sequential consistency across threads and the safest

7 References

Calciu, I., & Herlihy, M. (2011). *xchg, xadd, and cmpxchg instruction execution time in processor cycles* [Table]. In ResearchGate. https://www.researchgate.net/figure/xchg-xadd-and-cmpxchg-instruction-execution-time-in-processor-cycles_tbl12_224261171

Dürr, R. (n.d.). *A lock-free stack: A hazard pointer implementation explained (I)*. ModernesCpp.com. Retrieved from https://www.modernescpp.com/index.php/a-lock-free-stack-a-hazard-pointer-implementation-explained-i/

EPFL Distributed Computing Lab. (2020). *Memory Reclamation – Lecture 8* [PDF]. https://dcl.epfl.ch/site/_media/education/ca20_lecturemr.pdf

Hyeontaek Lim. (2015, November 6). *Memory Reclamation Techniques - CS 520 Lecture* [Video]. YouTube. https://www.youtube.com/watch?v=lsy8RRq2hHM

Overland, A. (n.d.). *Learn C++*. LearnCpp.com. Retrieved from https://www.learncpp.com/

GNU. (n.d.). *gprof-2.9.1 Manual*. Retrieved from https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

Valgrind. (n.d.). *Valgrind Quick Start Guide*. Retrieved from https://valgrind.org/docs/manual/quick-start.html#:~:text=1.,please%20read%20the%20User%20Manual.

University of Illinois Urbana-Champaign. (2022). *Stack vs Heap*. CS 225: Data Structures. Retrieved from https://courses.grainger.illinois.edu/cs225/fa2022/resources/stack-heap/

Disclaimer

The information, trading strategies, and materials presented in this report are provided strictly for educational and informational purposes and are offered without any guarantees or warranties regarding their accuracy, completeness, reliability, or timeliness. These materials are not intended to serve as financial, legal, tax, investment, or other professional advice, and no content herein should be interpreted as a recommendation to buy, sell, or hold any security, financial product, or instrument, nor as an endorsement of any specific strategy, practice, or course of action. Users of this material are strongly encouraged to conduct their own independent research, analysis, and due diligence or seek advice from qualified professionals before making any financial or investment decisions.

The authors, contributors, and distributors of this material are not acting as fiduciaries and do not assume any legal or ethical duty to the user. No advisory or client relationship is created through the use of this material. Trading and investing, particularly in derivatives, options, futures, and other leveraged products, involve substantial risks. These risks may include, but are not limited to, the potential for significant financial losses, the loss of principal, and exposure to unpredictable market conditions, volatility, or adverse economic factors. Users should be aware that past performance is not indicative of future results, and reliance on historical data or strategies described herein may not lead to favorable outcomes. Additionally, users are solely responsible for ensuring their trading or investment activities comply with applicable laws, regulations, and tax obligations in their jurisdiction.

By accessing and using this material, users acknowledge and agree to the following terms:

1. Assumption of Risk: Users accept all responsibility for the risks inherent in any trading or investment activity based on the information or strategies presented in this material.
2. No Liability: To the fullest extent permitted by applicable law, the creators, contributors, and distributors of this content disclaim all liability for any losses, damages, claims, or expenses—whether direct, indirect, incidental, or consequential—arising from reliance on or use of this material. This includes, but is not limited to, lost profits, trading losses, or disruptions to financial plans.
3. Indemnification: Users agree to indemnify, defend, and hold harmless the creators, contributors, and distributors of this material from and against any claims, damages, liabilities, or expenses (including attorney's fees) arising out of the user's reliance on or use of the information provided.
4. Personal Responsibility: Users affirm that they have independently evaluated the risks associated with any trading or investment decision and agree to proceed at their own discretion and liability.

Furthermore, users should be aware that the creators, contributors, and distributors of this content retain ownership of all intellectual property rights associated with this material. Unauthorized reproduction, distribution, or modification of this content is strictly prohibited and may violate applicable intellectual property laws. By accessing or downloading this content, users agree to abide by these terms and conditions and refrain from sharing, reselling, or otherwise redistributing this material without express written permission from its authors.

This material is provided on an "as is" basis and may include inaccuracies, typographical errors, or incomplete data. The authors reserve the right to make corrections or changes to the content at any time without notice. However, they are under no obligation to update, supplement, or revise this material to reflect changes in market conditions, new information, or other developments.

Users are strongly advised to consult with a licensed attorney, financial advisor, tax professional, or other qualified specialist to evaluate the implications of using the strategies or materials provided. Any reliance on the content of this material is entirely at the user's own risk. This disclaimer serves as a legally binding agreement between the user and the creators of this content, and users indicate their acceptance of these terms by accessing or utilizing this material in any form.

Lastly, trading and investing involve inherently speculative activities that may not be suitable for all individuals. The suitability of any particular investment strategy, asset, or trading approach depends on the user's specific financial situation, risk tolerance, objectives, and experience. Users should exercise caution and consider seeking professional guidance when engaging in financial activities that expose them to significant risks.