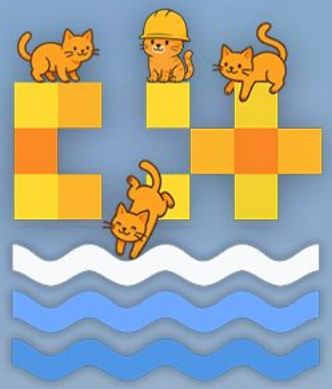


Beyond Sequential Consistency

Christopher Fretz

2025

Beyond Sequential Consistency

Bloomberg

Engineering

C++ On Sea 2025
June 23rd, 2025

Christopher Fretz
Senior C++ Engineer

TechAtBloomberg.com

Overview for this Talk

- The goal is to discuss the different memory ordering models available in C++, along with performance differences of those models.
- Starts with a quick introduction, defining terms and core concepts, before considering implementations and different memory models.
- Considers an elegant lock-free data structure (queue) that is extremely favorable for memory ordering optimization.
- Compares the relative performance of different memory models for this queue.
- Considers surprising outcomes where atomics are not necessary.

Disclaimers

- Lock-free programming is technically cool, but leads to complex solutions
- Hard to get right, hard to maintain, hard to test
- Often overused (do you really need it?)
- Benchmark, benchmark, benchmark
- Performance is often usage and hardware specific
- This talk focuses on performance of the data structure itself, and chooses a data structure that optimizes well. Your mileage may vary.

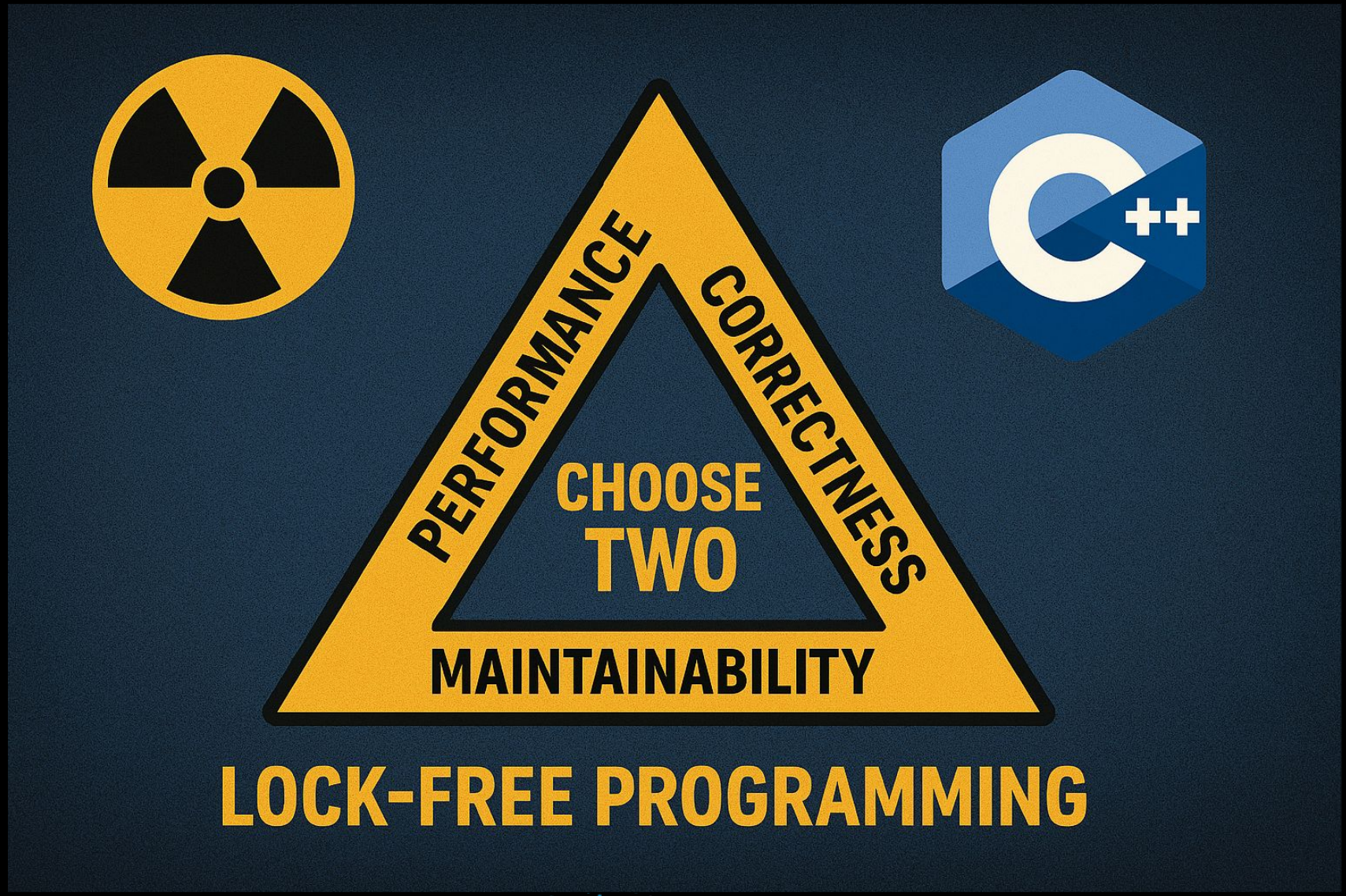


Image generated with ChatGPT 4o Image Gen

What Do We Mean by Lock-Free Programming?

- Colloquially, avoiding the use of mutexes and system calls to synchronize.
- Avoids software locks. Synchronization is still necessary.
- Depends on hardware intrinsics that guarantee certain operations are “uninterruptible” or “atomic”.
- More formally, lock freedom guarantees that at least one thread *always* makes progress.

What Do We Mean by Lock-Free Programming?

```cpp
#include <mutex>

std::mutex m;
int counter {0};

// Bad, but illustrative code
void increment(int val) {
    std::unique_lock l(m);
    counter += val;
}
```

What Do We Mean by Lock-Free Programming?

```cpp
#include <atomic>

std::atomic<int> counter {0};

// Bad, but illustrative code
void increment(int val) {
    int prev = counter.load();
    while (!counter.compare_exchange_strong(prev, prev + val)) {
        // Back off, somebody else succeeded
    }
}
```

What Do We Mean by Lock-Free Programming?

```
;; Traditional locked version
increment(int):
        push    rbx
        mov     ebx, edi
        mov     edi, OFFSET FLAT:m
        call    pthread_mutex_lock
        test    eax, eax
        jne     .L5
        add     DWORD PTR counter[rip], ebx
        mov     edi, OFFSET FLAT:m
        pop     rbx
        jmp     pthread_mutex_unlock
.L5:
        mov     edi, eax
        call    std::__throw_system_error(int)
```

```
;; Lock-free version
increment(int):
        mov     eax, DWORD PTR counter[rip]
.L2:
        lea     edx, [rdi+rax]
        lock cmpxchg    DWORD PTR counter[rip], edx
        jne     .L2
        ret
```

What Do We Mean by Lock-Free Programming?

```cpp
#include <atomic>

std::atomic<int> counter {0};

// Better version
void increment(int val) {
    counter += val;
}
```

```
increment(int):
        lock add        DWORD PTR counter[rip], edi
        ret
```

C++ Memory Model

- Starting in C++11, we gained a formally defined memory model that includes multiple different “memory orderings”:
  - Relaxed
  - Acquire
  - Release
  - Acq/Rel
  - Sequentially Consistent

C++ Memory Model

- Why all these different orderings? What is this for?
  - In a modern SMP machine, all cores are interacting with memory simultaneously
  - Each core has its own cache, meaning different cores may have independent copies of the same data
  - If operating on shared data, who sees which changes and at what time?
  - Do different cores even agree on the relative ordering of events?

C++ Memory Model

- Why all these different orderings? What is this for?
  - The different orderings of the C++ memory model give fine grained control over this, and bring order to chaos
  - C++ defines an abstract memory model with several available orderings that give different guarantees about how different threads (cores) will observe events
  - Vendors and library authors leverage hardware-specific intrinsics to implement and uphold this model

“Fences” and “Barriers”

- Modern machines and toolchains can reorder the execution of your code at build, link, and run time.
- Optimizer may decide to reorder logically independent operations under the “as-if” rule.
- CPU pipeline may decide to reorder instructions based on runtime data dependencies.
- Cache coherency protocol (load/store buffers) may cause loads and stores to be observed in different orders by different threads/cores. Runtime order can appear to violate common-sense causality.

“Fences” and “Barriers”

- “Memory fences” or “barriers” are the lowest order primitive used to constrain these reorderings.
- Multiple types of fences:
  - Compiler fence: exists purely for the build toolchain, does not emit instructions. Prevents toolchain from reordering operations.
  - Runtime fence: runtime opcodes to constrain the CPU pipeline/memory subsystem. Prevents the hardware from reordering operations.

“Fences” and “Barriers”

- “Memory fences” or “barriers” are the lowest order primitive used to constrain these reorderings.
- Multiple types of fences:
  - Store fence: Forbids reordering of memory stores across the fence.
  - Load fence: Forbids reordered of memory loads across the fence.
  - Combined/full fence: Forbids any reordering across the fence.

Sequential Consistency

Image generated with ChatGPT 4o Image Gen

C++ Memory Model

- Sequential Consistency
  - There exists a total ordering of all sequentially consistent operations that *all* threads agree upon.
  - Transitive visibility of operations is preserved; program appears to uphold expected causal relationships around all seq/cst operations, for all threads.
  - Default behavior.
  - Slowest behavior.

C++ Memory Model

```cpp
#include <atomic>
#include <thread>
#include <vector>

std::atomic<int> a {0}, b {0}, c {0};

// Operations are atomic, and all threads agree on the order
// in which the increments happen, even between separate variables.
int main() {
    std::vector<std::jthread> ts;
    ts.emplace_back([] { ++a; ++b; ++c; }); // each increment issues a full fence
    ts.emplace_back([] { ++b; ++c; ++a; }); // each increment issues a full fence
    ts.emplace_back([] { ++c; ++a; ++b; }); // each increment issues a full fence
}
```

Image generated with ChatGPT 4o Image Gen

C++ Memory Model

- Acquire/Release
  - Acquire/release usage on a specific atomic variable sets up a synchronization point between different threads.
  - If thread A stores into variable X with release ordering, and thread B loads that variable with acquire ordering, it is guaranteed that B can see all transitive stores that were visible to A at the time it made the store into X.
  - It is said that thread B “synchronizes with” thread A.
  - No guarantees are given for threads that don’t synchronize in this way.

C++ Memory Model

```cpp
#include <atomic>
#include <cassert>

bool non_atomic = false;
std::atomic<bool> atomic {false};

void producer_thread() {
    non_atomic = true;
    atomic.store(true, std::memory_order::release); // Store fence. Store cannot move below this.
}
void consumer_thread() {
    if (atomic.load(std::memory_order::acquire)) { // Load fence. Load cannot move above this.
        assert(non_atomic); // Guaranteed if executed
    }
}
```

Image generated with ChatGPT 4o Image Gen

C++ Memory Model

- Relaxed
  - Only guarantees atomicity (no thread can see a partial write) and modification order
    - “Modification order” gives a total ordering of writes to that one variable
  - No guarantees whatsoever in terms of ordering or visibility with other reads/writes
  - Establishes neither a runtime fence nor a compiler barrier, and can therefore be reordered arbitrarily at compile or run time

C++ Memory Model

```cpp
#include <atomic>
#include <iostream>
std::atomic<int> one {0}, two {0};
void thread_a() {
    auto loadtwo = two.load(std::memory_order::relaxed);
    one.store(loadtwo, std::memory_order::relaxed);
    std::cout << "loadtwo is: " << loadtwo << std::endl;
}
void thread_b() {
    two.store(42, std::memory_order::relaxed);
    auto loadone = one.load(std::memory_order::relaxed);
    std::cout << "loadone is: " << loadone << std::endl;
}
// => loadone is: 42
// => loadtwo is: 42
```

C++ Memory Model

```cpp
#include <atomic>
#include <iostream>
std::atomic<int> one {0}, two {0};
void thread_b() {
    two.store(42, std::memory_order::relaxed);
    auto loadone = one.load(std::memory_order::relaxed);
    std::cout << "loadone is: " << loadone << std::endl;
}
void thread_a() {
    auto loadtwo = two.load(std::memory_order::relaxed);
    one.store(loadtwo, std::memory_order::relaxed);
    std::cout << "loadtwo is: " << loadtwo << std::endl;
}
// => loadone is: 42
// => loadtwo is: 42
```

C++ Memory Model

- Relaxed
  - Gives the implementation total freedom to reoder at any (or multiple) level.
  - Gives the programmer very little ability to reason about causal relationships.

A Practical Lock-Free Data Structure

- “Prototypical example” that is also extraordinarily useful, elegant, and fast
- A lock-free ring buffer uses a circular buffer of memory, along with atomic counters, to move data between threads
- Basic use case is single producer/single consumer; can be trivially extended to single producer/multiple consumer

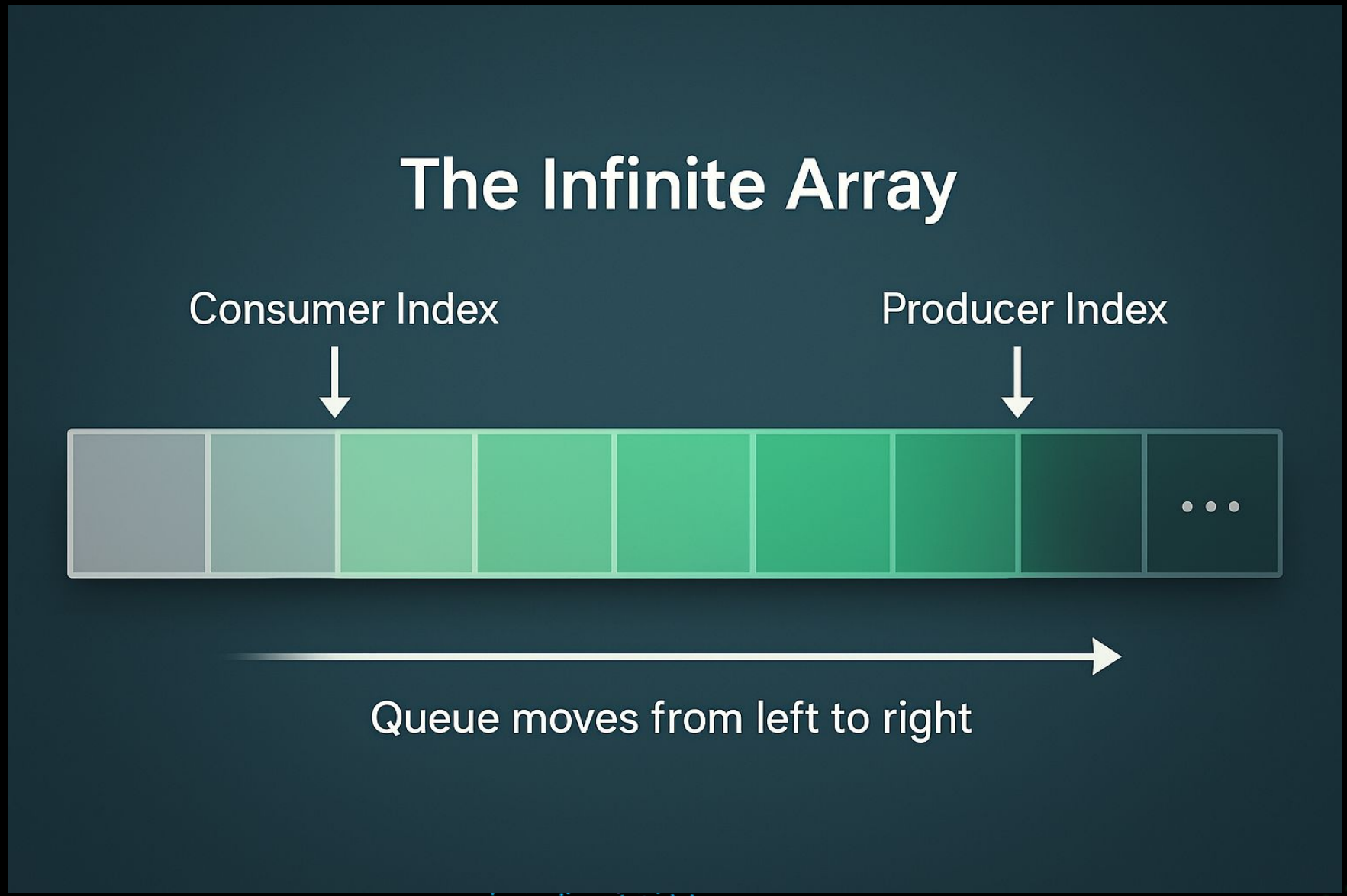


Image generated with ChatGPT 4o Image Gen

A Practical Lock-Free Data Structure

- In the “infinite array” concept, a queue is trivially implemented using two atomic counters: a producer index and a consumer index
- Enqueues are performed by copying data into the array and incrementing the producer index
- Dequeues are performed by copying data out of the array and incrementing the consumer index
- Indexes increase monotonically

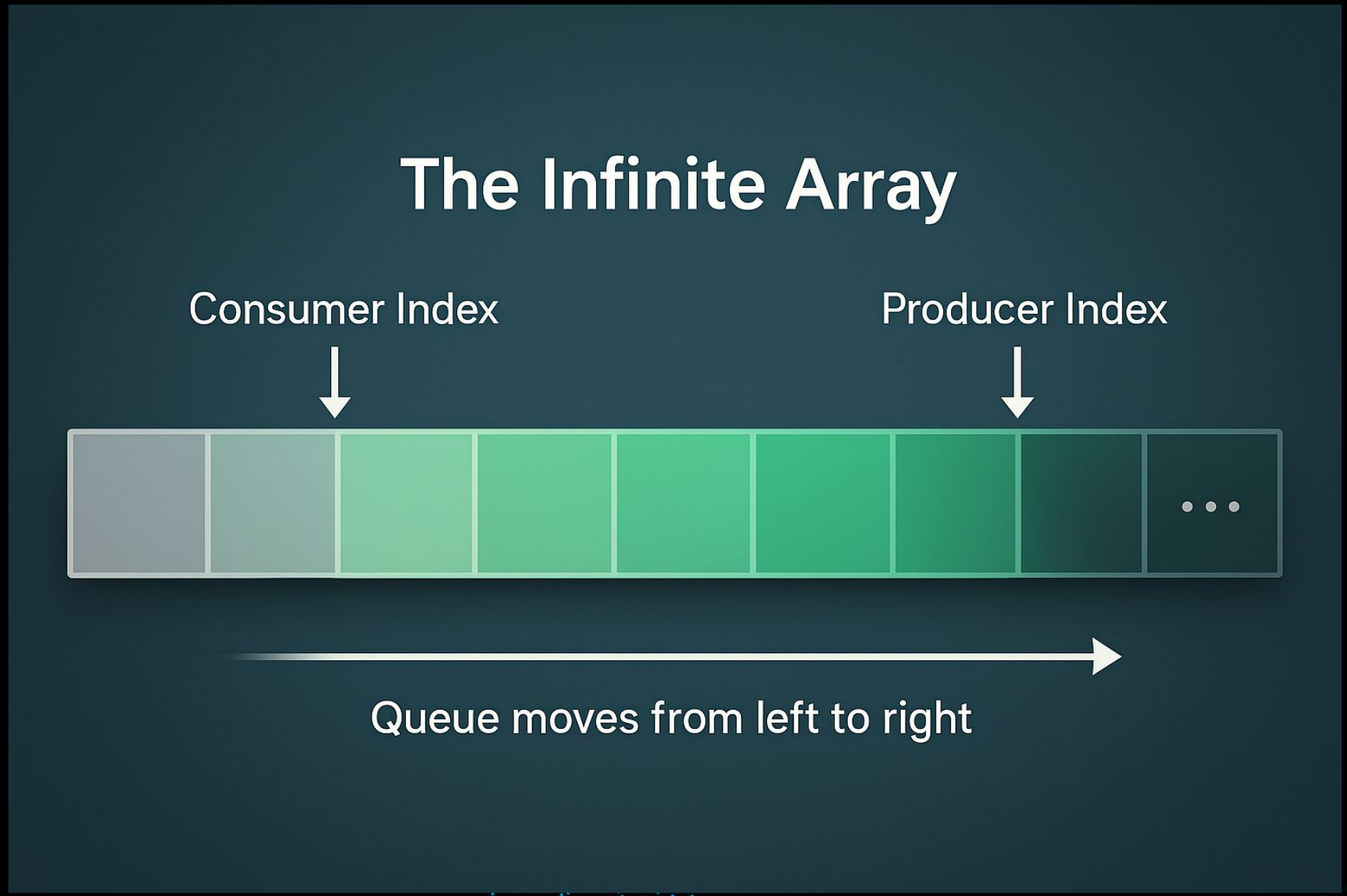


Image generated with ChatGPT 4o Image Gen

A Practical Lock-Free Data Structure

- In practice, we only have access to finite memory, but the design can be trivially adapted
- Queue maintains two atomic counters: a producer index and a consumer index
- Counters are indexes into the “infinite” array, but modular arithmetic maps them into a finite buffer
- Buffer is used circularly, wrapping back around to the beginning from the end

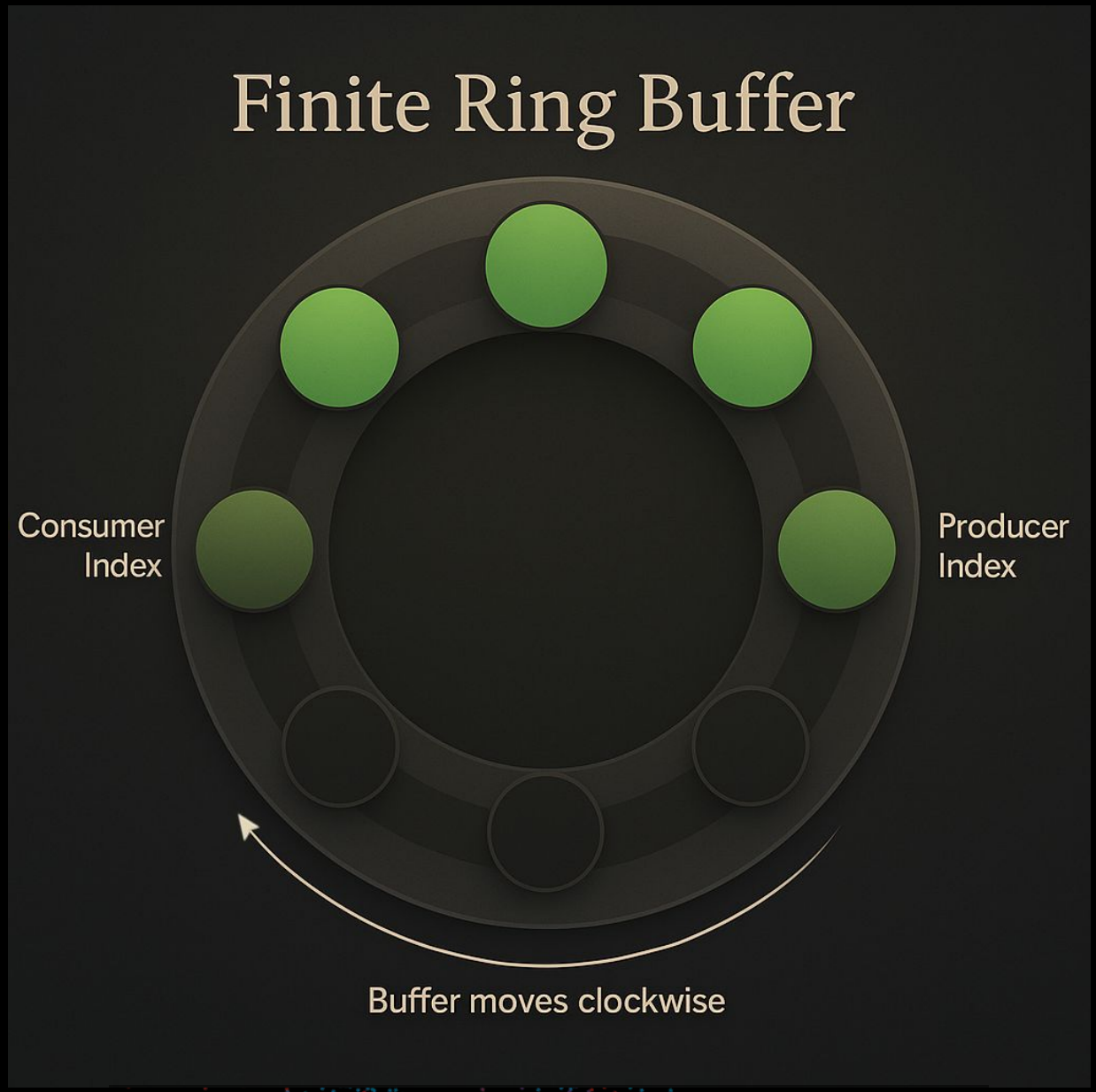


Image generated with ChatGPT 4o Image Gen

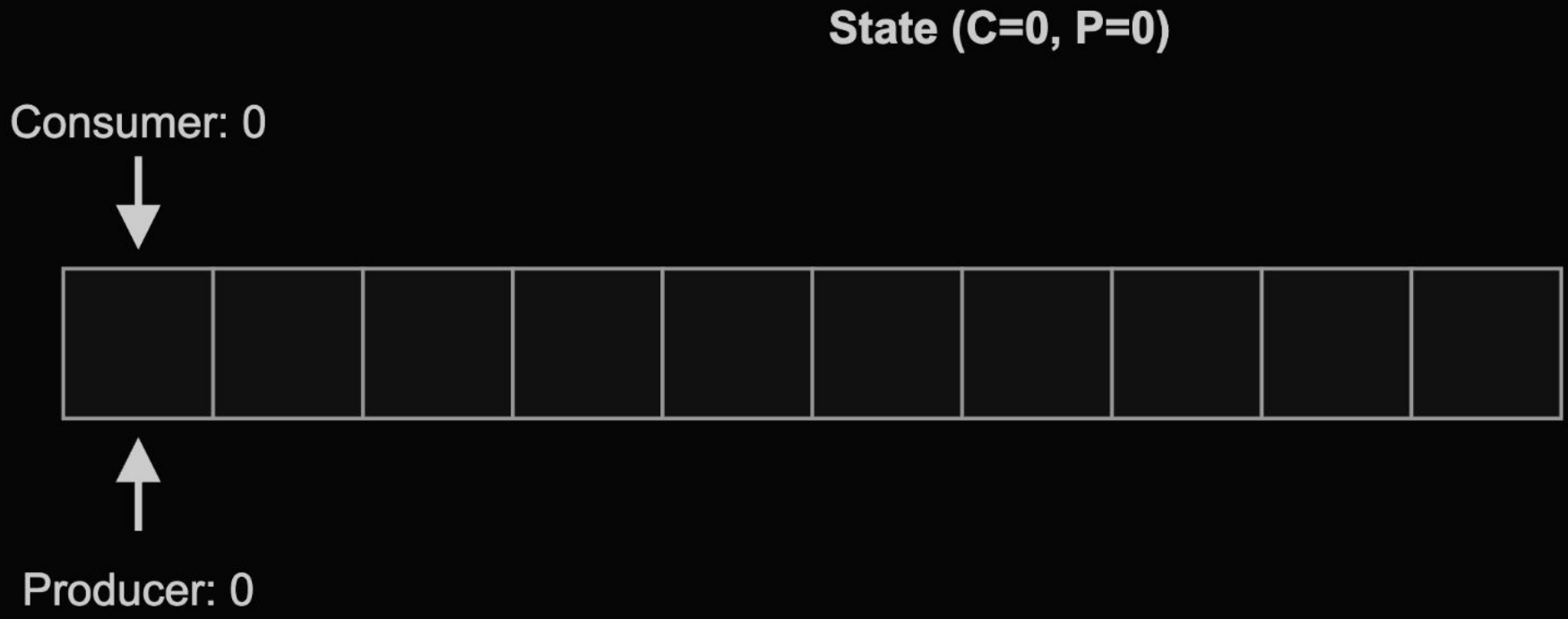


Image generated with ChatGPT 4o Image Gen

Image generated with ChatGPT 4o Image Gen

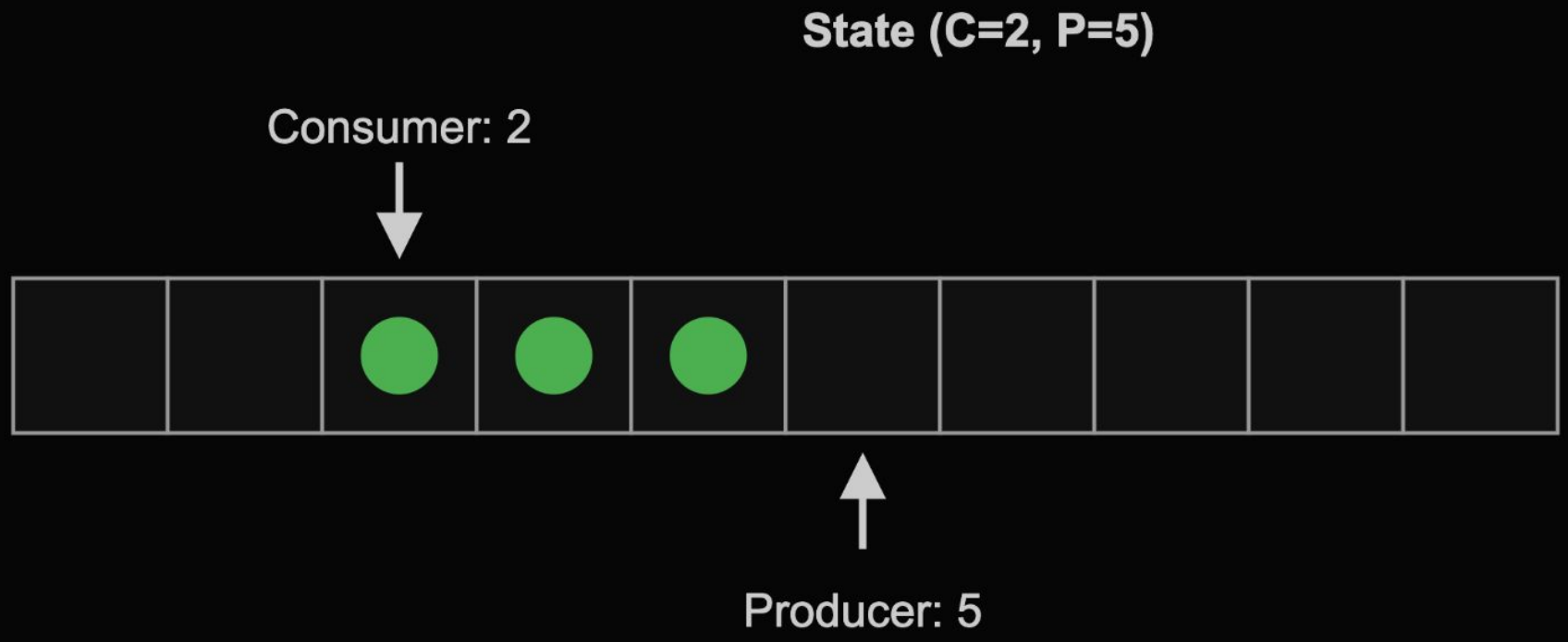


Image generated with ChatGPT 4o Image Gen

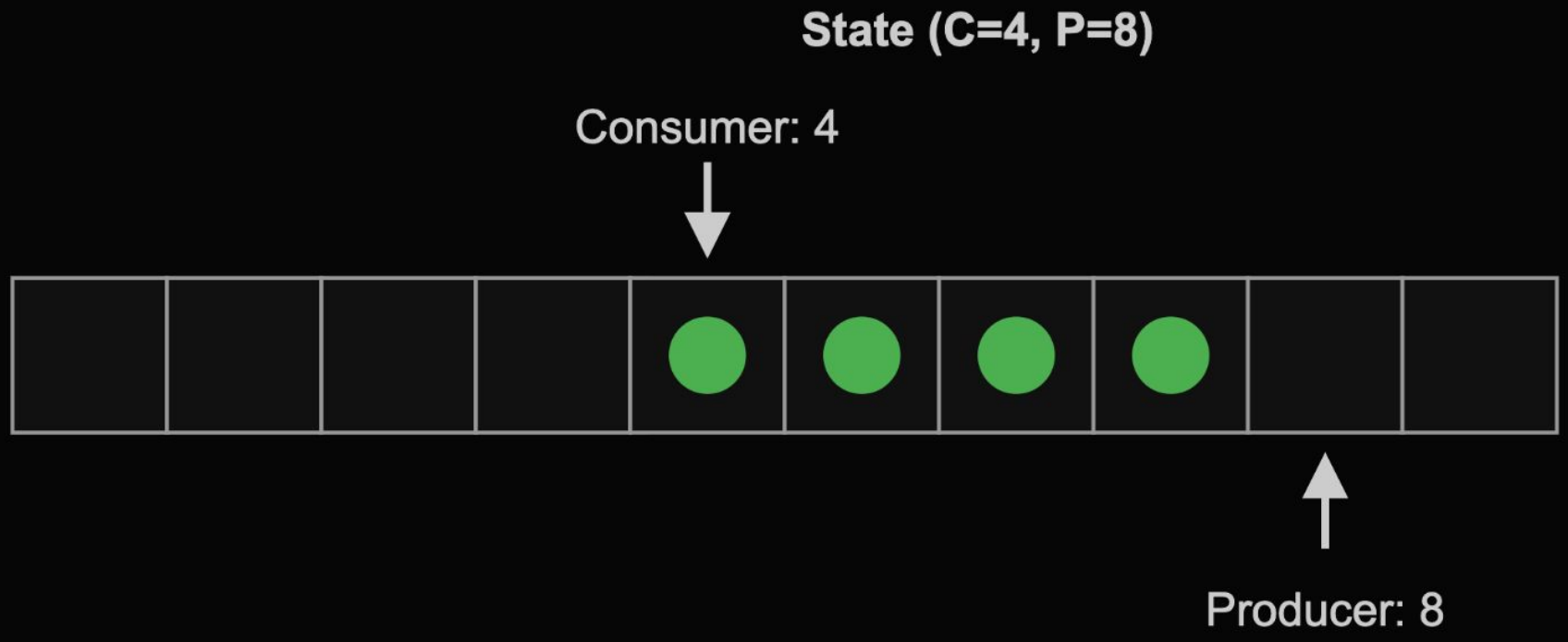


Image generated with ChatGPT 4o Image Gen

Image generated with ChatGPT 4o Image Gen

A Practical Lock-Free Data Structure

```
void producer(ring_buffer* queue) {
    int64_t counter = 0;
    while (counter < 10'000'000'000) {
        queue->push(counter);
        ++counter;
    }
}
void consumer(ring_buffer* queue) {
    int64_t counter = 0;
    while (counter < 10'000'000'000) {
        auto val = queue->pop();
        assert(val == counter);
        ++counter;
    }
}
```

A Practical Lock-Free Data Structure

```
struct ring_buffer { // Would be a class template in real code
    static constexpr std::size_t capacity = 32'768;

    ring_buffer() : next_producer_record(0), next_consumer_record(0) {}
    inline void push(int64_t record) noexcept;
    inline int64_t pop() noexcept;

    inline std::size_t size() const noexcept;
    inline void sleep(std::size_t& spin_counter) const noexcept;
    inline std::size_t map_index(std::size_t index) const noexcept;

    std::atomic<std::size_t> next_producer_record;
    std::atomic<std::size_t> next_consumer_record;
    std::array<int64_t, capacity> data;
};
```

A Practical Lock-Free Data Structure

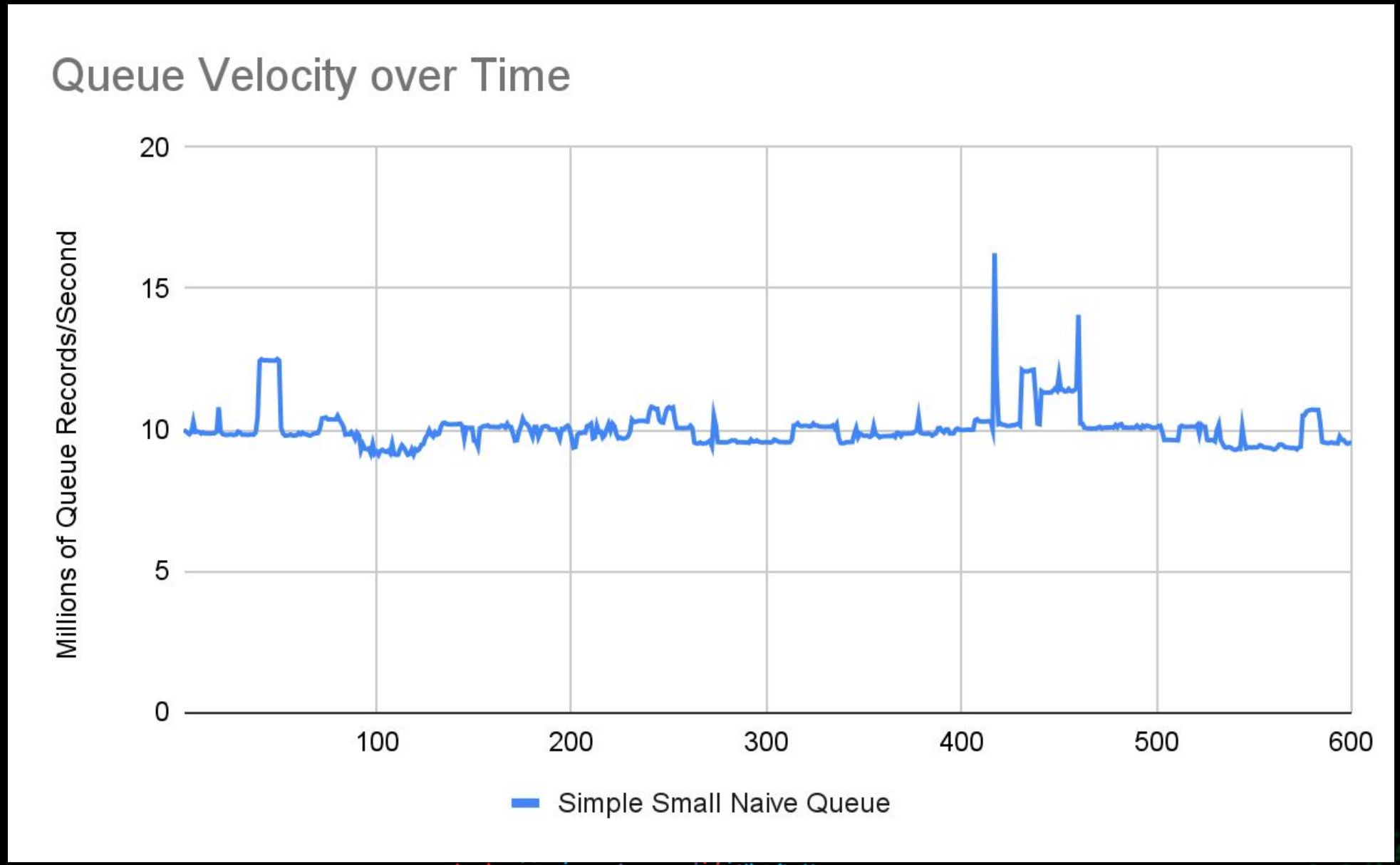
```
inline void ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    while (size() == capacity) sleep(spin_counter);

    // Produce the record
    data[map_index(next_producer_record)] = record;
    ++next_producer_record;
}
inline std::size_t ring_buffer::size() const noexcept {
    return next_producer_record - next_consumer_record;
}
inline std::size_t ring_buffer::map_index(std::size_t index) const noexcept {
    return index % capacity; // Could be a bitmask
}
```

A Practical Lock-Free Data Structure

```
inline int64_t ring_buffer::pop() noexcept {
    // Wait until there's data
    std::size_t spin_counter = 0;
    while (!size()) sleep(spin_counter);

    // Consume the record
    auto record = data[map_index(next_consumer_record)];
    ++next_consumer_record;
    return record;
}
inline void ring_buffer::sleep(std::size_t& spin_counter) const noexcept {
    using namespace std::chrono_literals;
    if (spin_counter++ < 16) return;
    else std::this_thread::sleep_for(50us);
}
```

A Practical Lock-Free Data Structure

```
void print_stats(ring_buffer const* queue) {
    std::size_t prev_idx = 0;
    while (runflag) {
        std::size_t new_idx = queue->next_consumer_record;
        std::cout << "Queue is moving at " << (new_idx - prev_idx) << " records/second." << std::endl;
        std::this_thread::sleep_for(1s);
        prev_idx = new_idx;
    }
}
int main() {
    auto queue = std::make_unique<ring_buffer>();
    std::thread stats {print_stats, queue.get()}, prod {producer, queue.get()}, cons {consumer, queue.get()};
    prod.join(); cons.join();
    runflag = false; stats.join();
}
```


Queue Velocity over Time
Millions of Queue Records/Second
20
15
10
5
0
100
200
300
400
500
600
Simple Small Naive Queue

Performance Analysis

- Roughly ten million queue records per second might sound like a lot, but it's **really not**
- At 2.6 Ghz, 10 million records/second means we're at ~260 cycles per record
- Our queue is doing **absolutely no work**, and it's moving **integers**
- We should be able to go **substantially** faster
- What gives?

Performance Analysis

```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    while (size() == capacity)
        sleep(spin_counter);

    // Produce the record
    data[map_index(next_producer_record)] = record;
    ++next_producer_record;
}
```

```
.L40:
        add     rbx, 1                  ;; incr spin count
.L45:
        mov     rax, QWORD PTR [rbp+0] ;; producer index load
        mov     rdx, QWORD PTR [r12]   ;; consumer index load
        sub     rax, rdx                ;; get queue size
        cmp     rax, 32768              ;; queue capacity check
        jne     .L48                    ;; space in the queue
        cmp     rbx, 15                 ;; spinlock counter check
        jbe     .L40                    ;; keep spinning
;; ...setup for nanosleep...
.L42:
        call    nanosleep               ;; backoff
```

Performance Analysis

```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    while (size() == capacity)
        sleep(spin_counter);

    // Produce the record
    data[map_index(next_producer_record)] = record;
    ++next_producer_record;
}
```

```
.L48:
        mov     rax, QWORD PTR [rbp+0]          ;; prod index load
        and     eax, 32767                      ;; map queue index
        mov     QWORD PTR [rbp+16+rax*8], r13 ;; store in queue
        lock add        QWORD PTR [rbp+0], 1    ;; incr prod index
        add     r13, 1                          ;; incr next record
```

Performance Analysis

```
inline int64_t ring_buffer::pop() noexcept {
    // Wait until there's data
    std::size_t spin_counter = 0;
    while (!size()) sleep(spin_counter);

    // Consume the record
    auto record = data[map_index(next_consumer_record)];
    ++next_consumer_record;
    return record;
}
```

```
.L13:
        add     rbx, 1                  ;; incr spin count
.L19:
        mov     rdx, QWORD PTR [r12]    ;; producer index load
        mov     rax, QWORD PTR [rbp+0]  ;; consumer index load
        cmp     rdx, rax                ;; is data in queue
        jne     .L22                    ;; data is in queue
        cmp     rbx, 15                 ;; spinlock counter check
        jbe     .L13                    ;; keep spinning
;; ...setup for nanosleep...
.L15:
        call    nanosleep               ;; backoff
```

Performance Analysis

```
inline int64_t ring_buffer::pop() noexcept {
    // Wait until there's data
    std::size_t spin_counter = 0;
    while (!size()) sleep(spin_counter);

    // Consume the record
    auto record = data[map_index(next_consumer_record)];
    ++next_consumer_record;
    return record;
}
```

```
.L22:
mov     rax, QWORD PTR [rbp+0]          ;; cons index load
and     eax, 32767                      ;; map queue index
mov     rax, QWORD PTR [r12+16+rax*8]   ;; load rec from q
lock add        QWORD PTR [rbp+0], 1    ;; incr cons index
cmp     rax, r13                        ;; assertion check
jne     .L23                            ;; failed assertion
add     r13, 1                          ;; predict next rec
```

Performance Analysis

- The only potential bottleneck in the assembly is the `lock add` instruction
- This is a read-modify-write instruction and issues a full memory fence
- Do we really need this?

The Cost of Operators

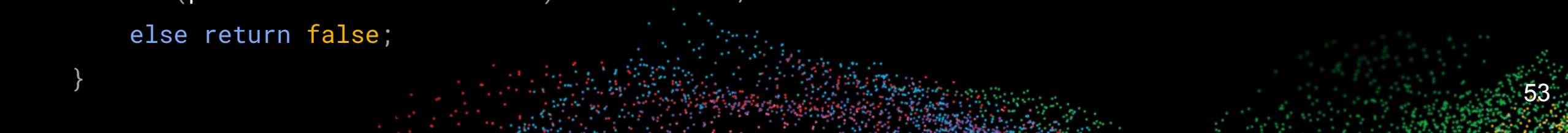
- By using the default API on `std::atomic` (operators), we are implicitly requesting full sequential consistency.
- By using the pre-increment operator, we are implicitly requesting the read-modify-write (fetch add).
- The producer and consumer are individually single threaded, and our access pattern doesn’t require sequential consistency.
- We can avoid both.

Atomic Optimization

```
inline void ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer = next_producer_record.load(std::memory_order::acquire);
    while (!has_space(producer)) sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    next_producer_record.store(producer + 1, std::memory_order::release);
}
inline bool ring_buffer::has_space(std::size_t producer) const noexcept {
    std::size_t consumer = next_consumer_record.load(std::memory_order::acquire);
    if (producer - consumer < capacity) return true;
    else return false;
}
```

Atomic Optimization

```
inline int64_t ring_buffer::pop() noexcept {
    // Wait until there's data
    std::size_t spin_counter = 0;
    std::size_t consumer = next_consumer_record.load(std::memory_order::acquire);
    while (!has_data(consumer)) sleep(spin_counter);

    // Consume the record
    auto record = data[map_index(consumer)];
    next_consumer_record.store(consumer + 1, std::memory_order::release);
    return record;
}
inline bool ring_buffer::has_data(std::size_t consumer) const noexcept {
    std::size_t producer = next_producer_record.load(std::memory_order::acquire);
    if (producer - consumer != 0) return true;
    else return false;
}
```



Atomic Optimization

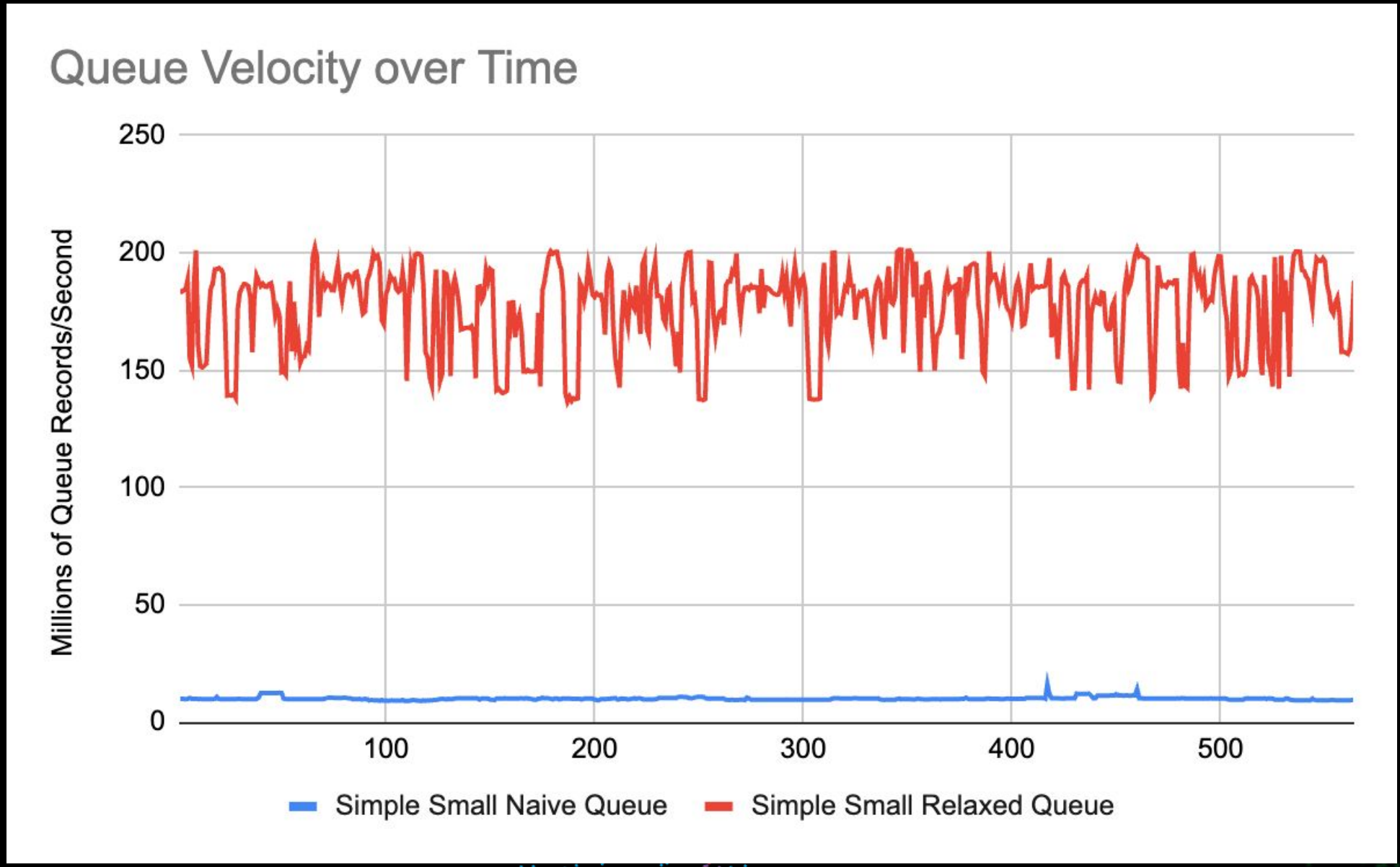
```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer =
       next_producer_record.load(
           std::memory_order::acquire);
    while (!has_space(producer))
        sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    next_producer_record.store(
        producer + 1, std::memory_order::release);
}
```

```
.L49:
        mov     rbx, QWORD PTR [r12]   ;; producer index load
        xor     r15d, r15d             ;; sleep counter reset
.L45:
        mov     rdx, QWORD PTR [r13+0] ;; consumer index load
        mov     rax, rbx
        sub     rax, rdx               ;; get records in queue
        cmp     rax, 32767             ;; queue capacity check
        jbe     .L59                   ;; space is available
        cmp     r15, 15                ;; spinlock counter check
        jbe     .L51                   ;; we are still spinning
;; ...setup for nanosleep call...
.L47:
        call    nanosleep              ;; backoff
;; ...error handling for nanosleep...
.L51:
        add     r15, 1                 ;; incr spin count
        jmp     .L45
```

Atomic Optimization

```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer =
       next_producer_record.load(
           std::memory_order::acquire);
    while (!has_space(producer))
        sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    next_producer_record.store(
        producer + 1, std::memory_order::release);
}
```

```
        jmp   .L49                          ;; entrypoint
.L59:
        mov   rax, rbx                      ;; backup prod index
        add   rbx, 1                        ;; incr prod index
        and   eax, 32767                    ;; map queue index
        mov   QWORD PTR [r12+16+rax*8], rbp ;; store rec in queue
        add   rbp, 1                        ;; incr current record
        mov   QWORD PTR [r12], rbx          ;; update prod index
        cmp   rbp, r14                      ;; completion check
        je    .L58                          ;; producer terminates
.L49:
        mov   rbx, QWORD PTR [r12]          ;; load producer index
        xor   r15d, r15d                    ;; from previous slide
```


Queue Velocity over Time
Millions of Queue Records/Second
250
200
150
100
50
0
100
200
300
400
500
Simple Small Naive Queue
Simple Small Relaxed Queue

Performance Analysis

- New code has *literally zero* atomic instructions or fences on x86_64
- At 2.6 Ghz, 170 million records/second is ~15 cycles per record
- Are these numbers *too* good?
- Test machine has 32K of L1 cache, 1 MB of L2 cache; Queue contains only integers: entire thing fits in L2, about ⅛ fits in L1
- Too small to be realistic, but shows what the hardware is capable of
- Can we still go faster?

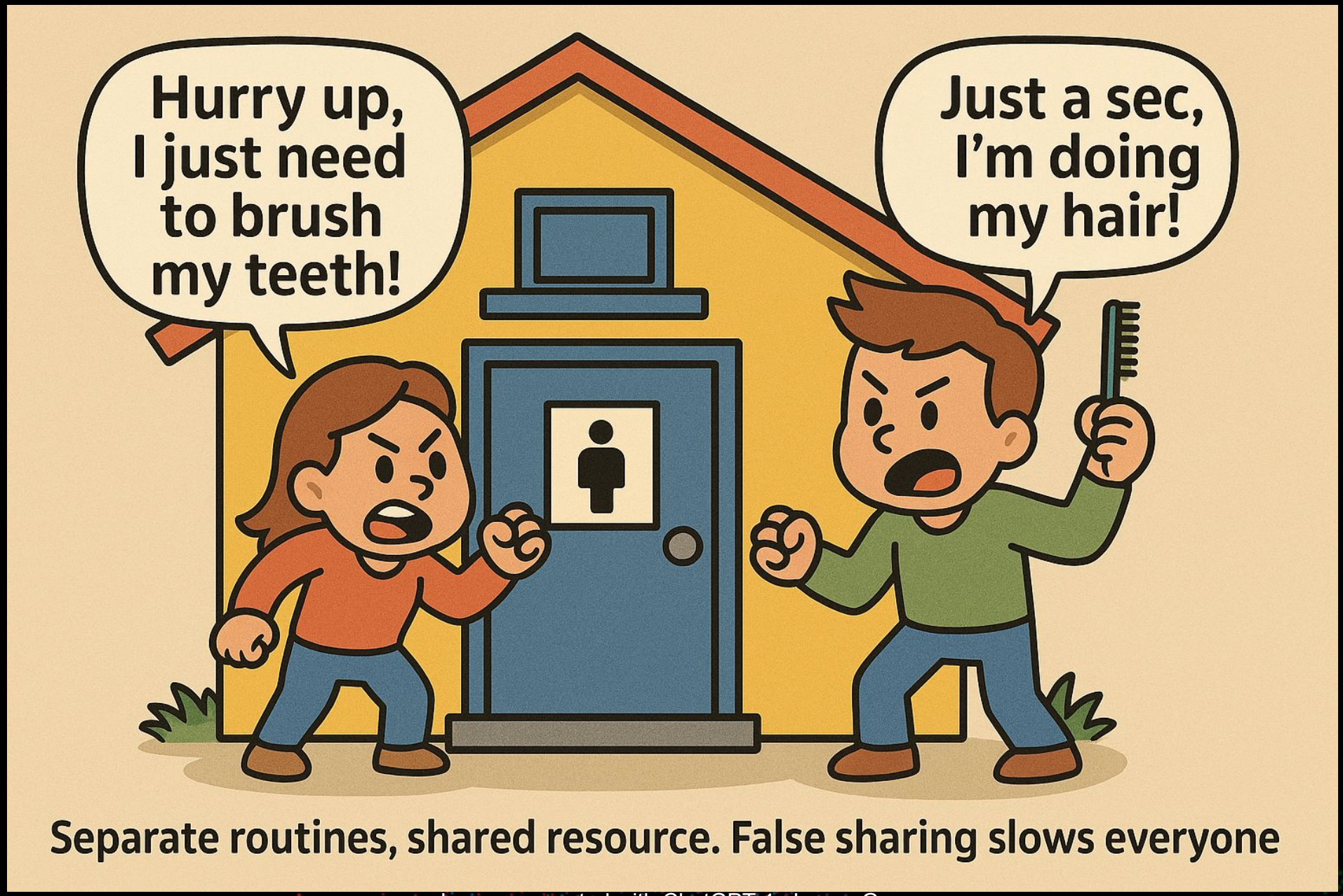


Image generated with ChatGPT 4o Image Gen

Next Bottleneck

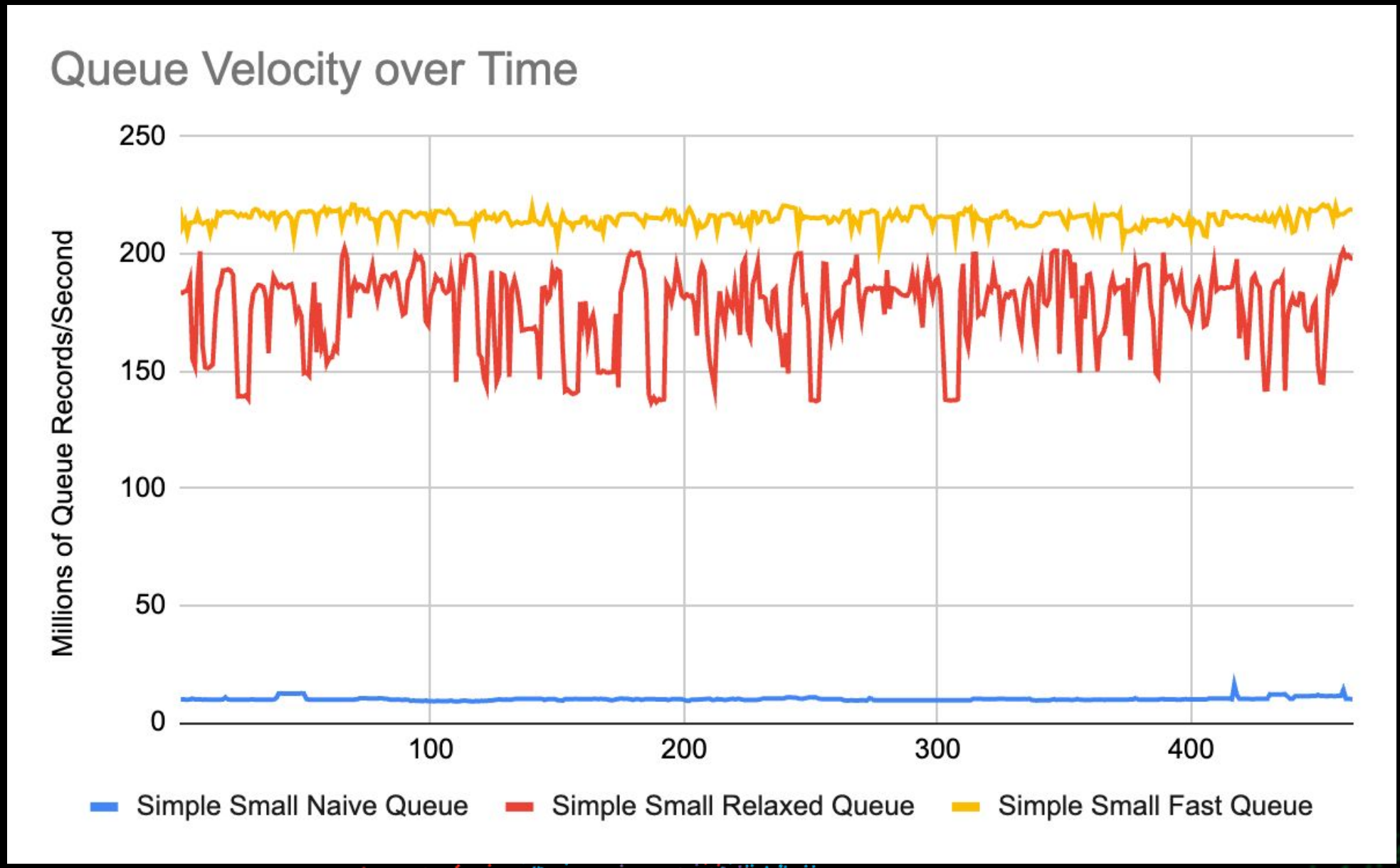
```
struct ring_buffer { // Would be a class template in real code
    static constexpr std::size_t capacity = 32'768;

    ring_buffer() : next_producer_record(0), next_consumer_record(0) {}
    inline void push(int64_t record) noexcept;
    inline int64_t pop() noexcept;

    inline std::size_t size() const noexcept;
    inline void sleep(std::size_t& spin_counter) const noexcept;
    inline std::size_t map_index(std::size_t index) const noexcept;

    std::atomic<std::size_t> next_producer_record; // sizeof(next_producer_record) == 8
    std::atomic<std::size_t> next_consumer_record; // both indexes on the same cache line
    std::array<int64_t, capacity> data;
};
```

Next Bottleneck

- Current memory layout packs the producer index and consumer index directly next to each other; they likely fall on the same cache line.
- Different cores will fight to modify the same line concurrently.
- Cache lines are the minimum granularity for most memory subsystems.
- MESI cache protocol will require bouncing the line back and forth.
- `std::hardware_destructive_interference_size` exists to solve this.

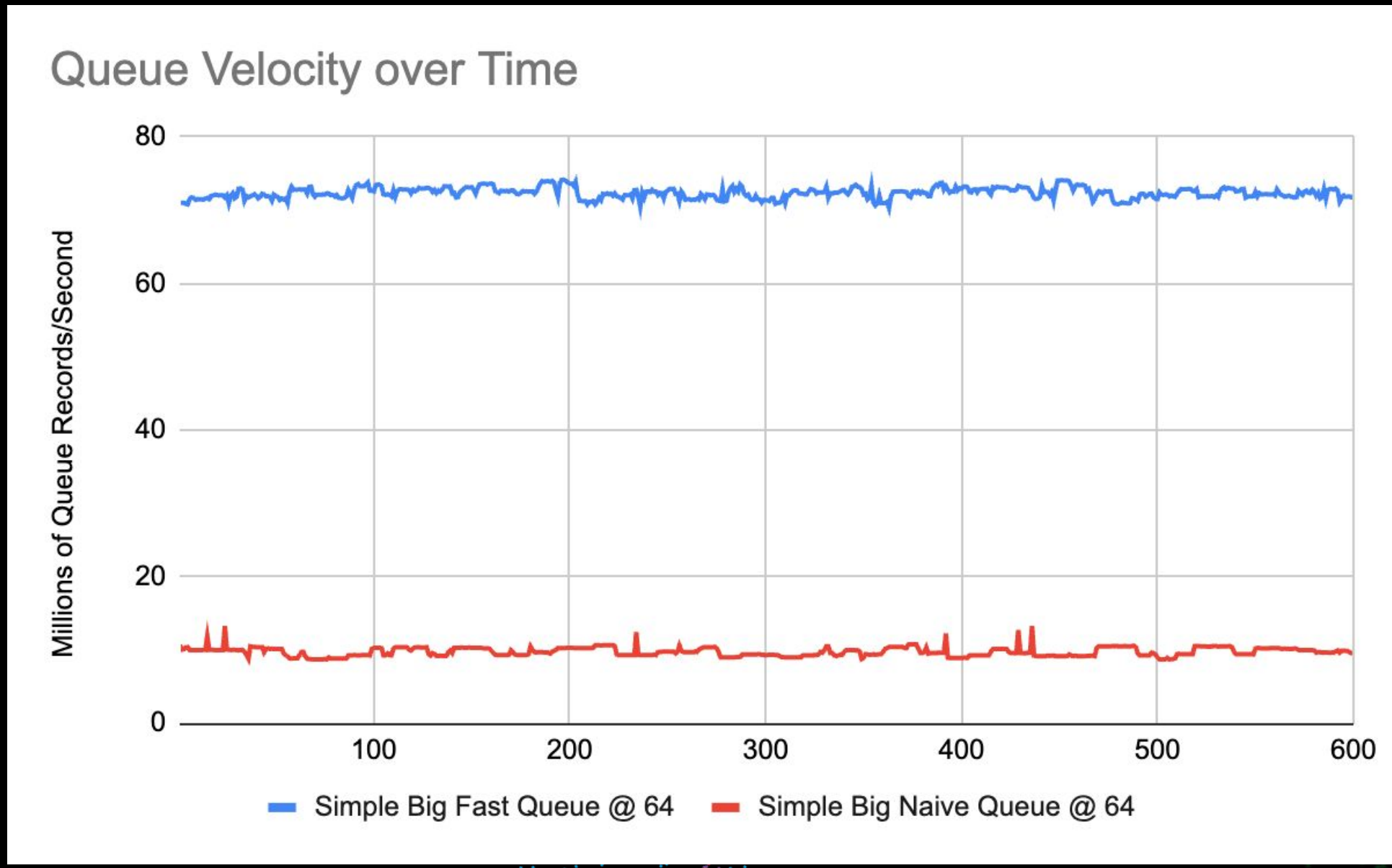
Next Bottleneck

```
struct ring_buffer { // Would be a class template in real code
    static constexpr std::size_t capacity = 32'768;
    static constexpr std::size_t cache_line_size = std::hardware_destructive_interference_size;

    ring_buffer() : next_producer_record(0), next_consumer_record(0) {}
    inline void push(int64_t record) noexcept;
    inline int64_t pop() noexcept;

    inline bool has_space(std::size_t producer) const noexcept;
    inline bool has_data(std::size_t consumer) const noexcept;
    inline void sleep(std::size_t& spin_counter) const noexcept;
    inline std::size_t map_index(std::size_t ptr) const noexcept;

    alignas(cache_line_size) std::atomic<std::size_t> next_producer_record; // record is now 64 bytes
    alignas(cache_line_size) std::atomic<std::size_t> next_consumer_record; // guaranteed unique lines
    alignas(cache_line_size) std::array<int64_t, capacity> data;
};
```

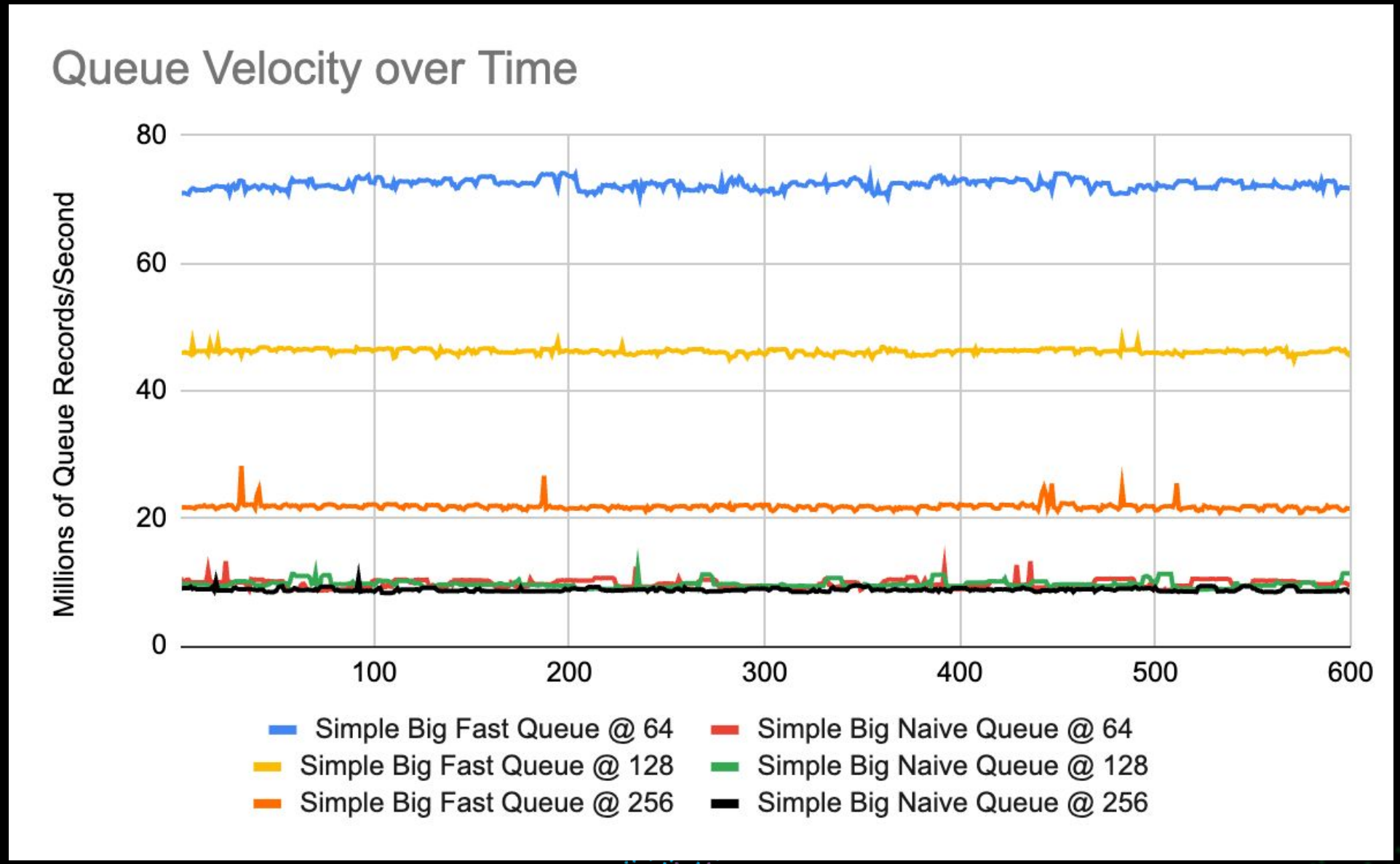

Queue Velocity over Time
250
200
150
100
50
0
Millions of Queue Records/Second
100
200
300
400
Simple Small Naive Queue
Simple Small Relaxed Queue
Simple Small Fast Queue

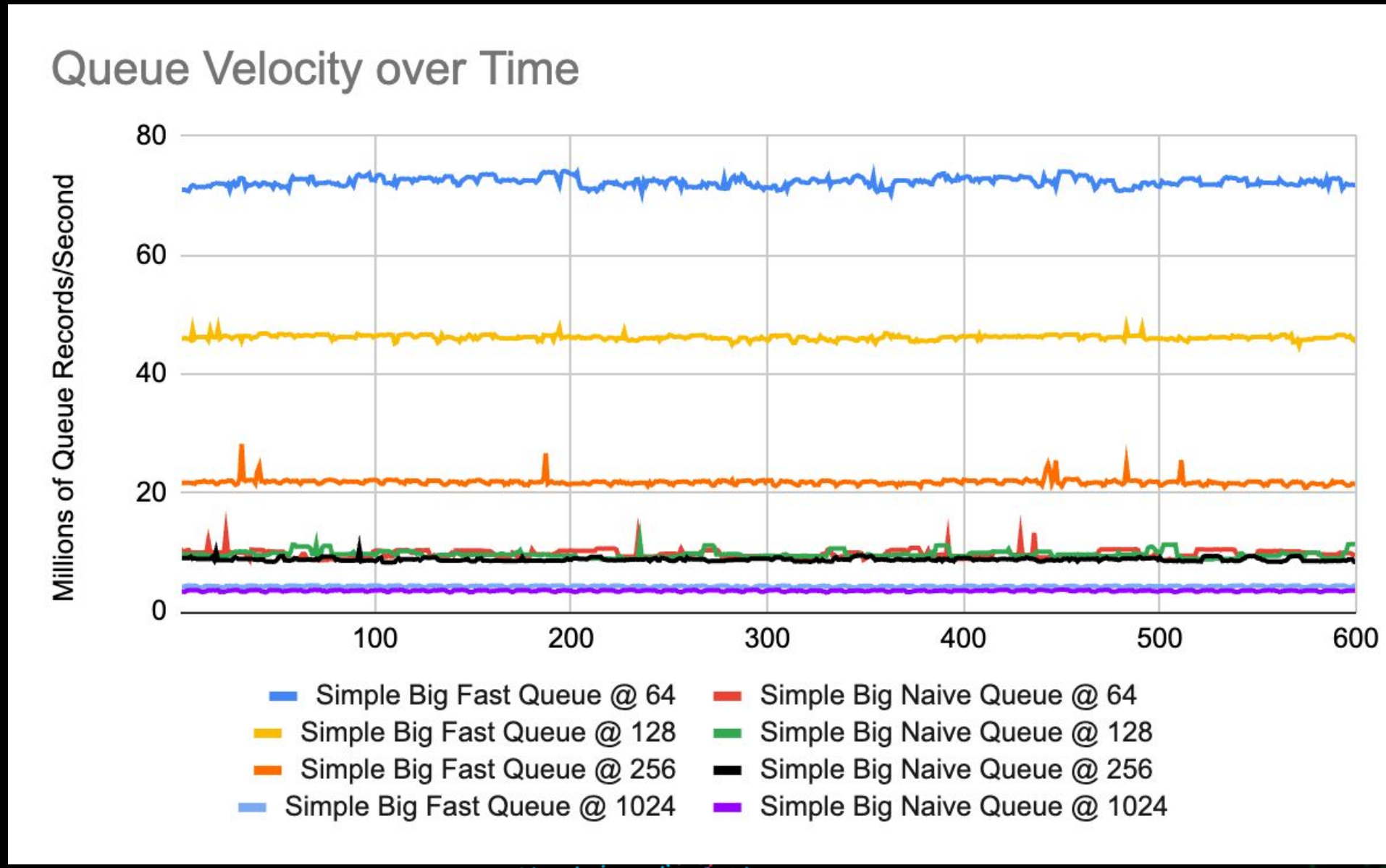
Next Steps

- HUGE difference (over 20x when aligned) using different memory orderings!
- Queue is too trivial to be realistic
- How can we make benchmark more realistic?
- First step: See how performance trends as data gets larger


Queue Velocity over Time
Millions of Queue Records/Second
80
60
40
20
0
100
200
300
400
500
600
Simple Big Fast Queue @ 64
Simple Big Naive Queue @ 64


Queue Velocity over Time
Millions of Queue Records/Second
80
60
40
20
0
100
200
300
400
500
600
Simple Big Fast Queue @ 64
Simple Big Fast Queue @ 128
Simple Big Naive Queue @ 64
Simple Big Naive Queue @ 128


Queue Velocity over Time
Millions of Queue Records/Second
80
60
40
20
0
100
200
300
400
500
600
Simple Big Fast Queue @ 64
Simple Big Fast Queue @ 128
Simple Big Fast Queue @ 256
Simple Big Naive Queue @ 64
Simple Big Naive Queue @ 128
Simple Big Naive Queue @ 256


Queue Velocity over Time
Millions of Queue Records/Second
80
60
40
20
0
100
200
300
400
500
600
Simple Big Fast Queue @ 64
Simple Big Fast Queue @ 128
Simple Big Fast Queue @ 256
Simple Big Fast Queue @ 1024
Simple Big Naive Queue @ 64
Simple Big Naive Queue @ 128
Simple Big Naive Queue @ 256
Simple Big Naive Queue @ 1024

Next Steps

- Even at substantially larger data sizes, performance diff is durable
- Have to go to 1024 bytes before copy is the dominating factor over synchronization
- Even in this case, we still save ~25%
- How else can we make this more realistic?

Multi-Consumer Broadcast Queue

- A traditional “work queue” divides incoming data between a pool of cooperating threads.
- In a broadcast queue, by contrast, *all* consumer threads receive *all* data.
- Each consumer maintains its own individual queue index.
- Useful in workflows like semantic analysis, intrusion detection, classification, etc, where many different things need to be done on each data record.
- Think of it like Kafka, except entirely in-memory and lock-free.

Multi-Consumer Broadcast Queue

- Multiple consumers adds a new problem, how does the producer know if the queue is full?
- Need to add a new thread, whose job is to identify the “slowest consumer”.
- Let’s call this new thread the “collector”.
- The “collector” continuously observes the consumer indexes and identifies the slowest consumer in the queue (lowest index), which it records as its own index, and becomes the new “edge” of the circular buffer.

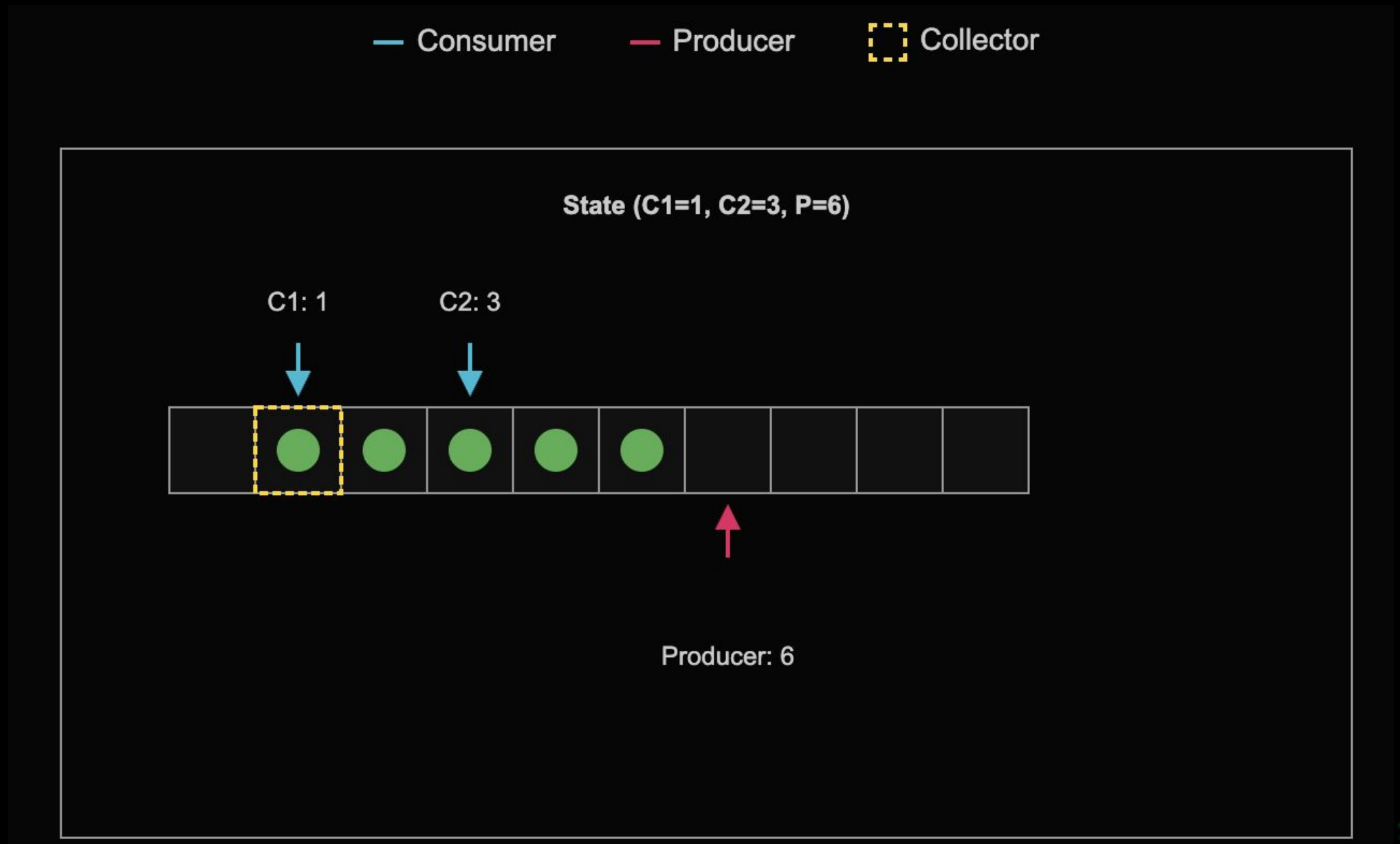


Image generated with ChatGPT 4o Image Gen

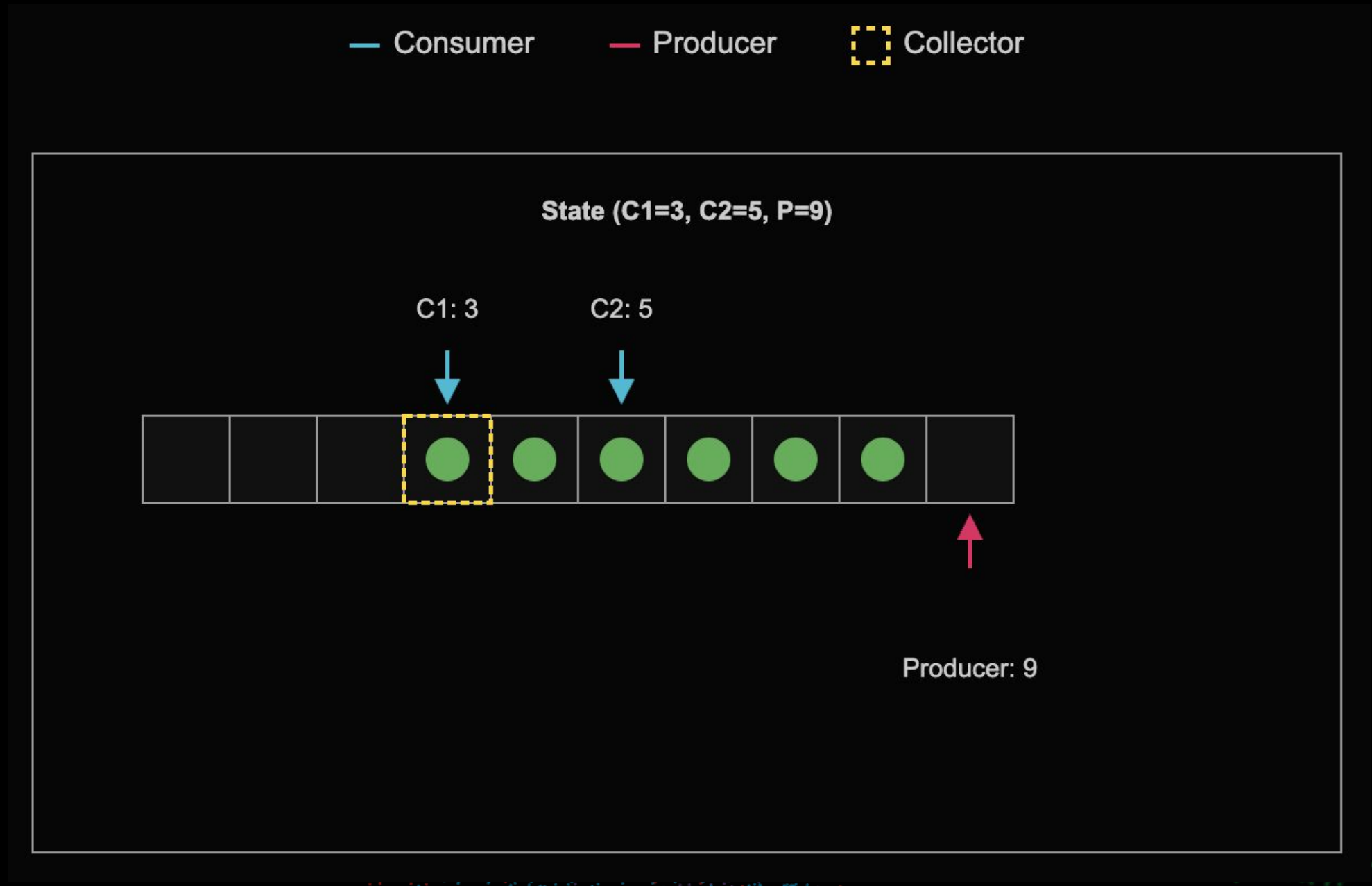


Image generated with ChatGPT 4o Image Gen

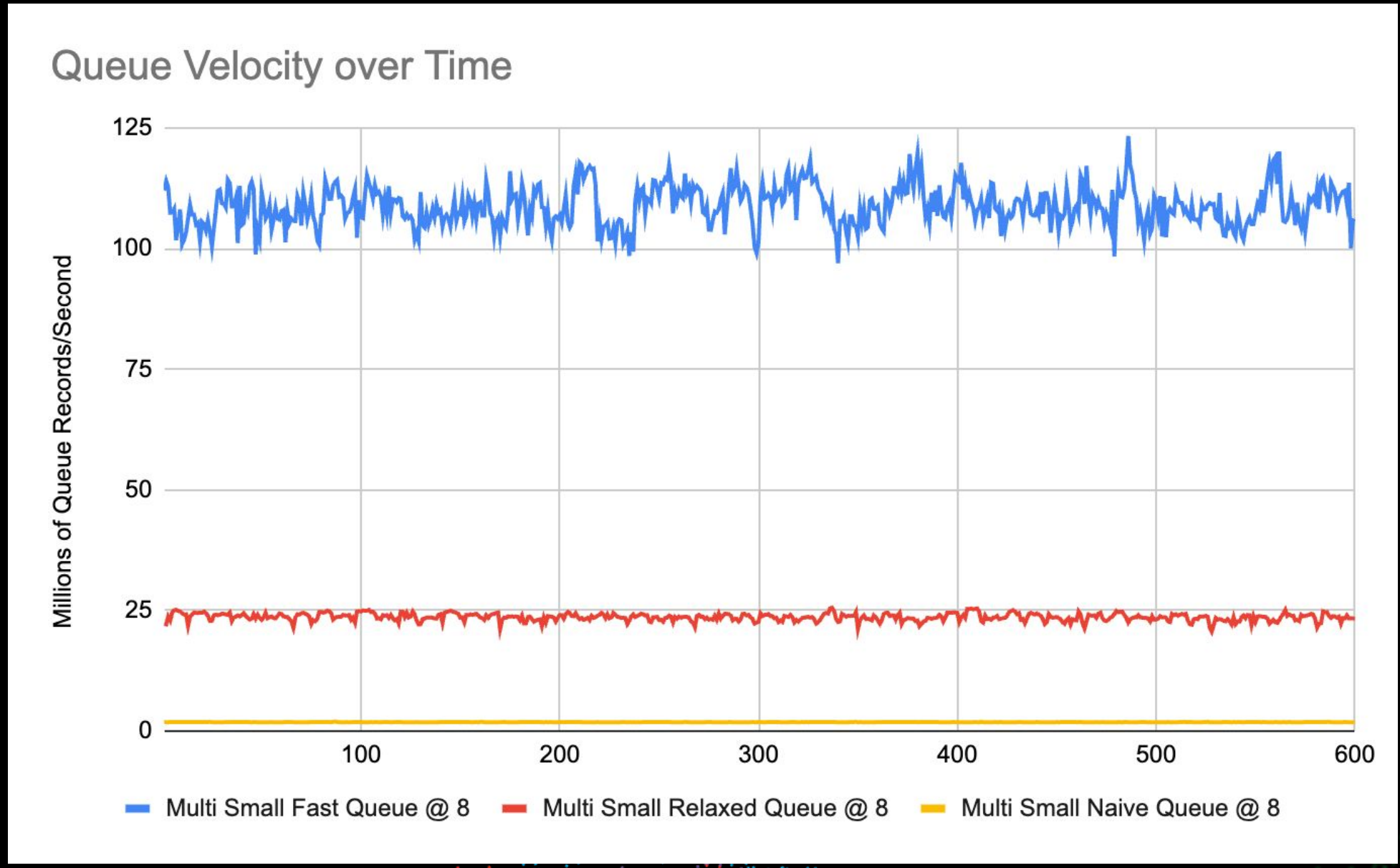
Image generated with ChatGPT 4o Image Gen

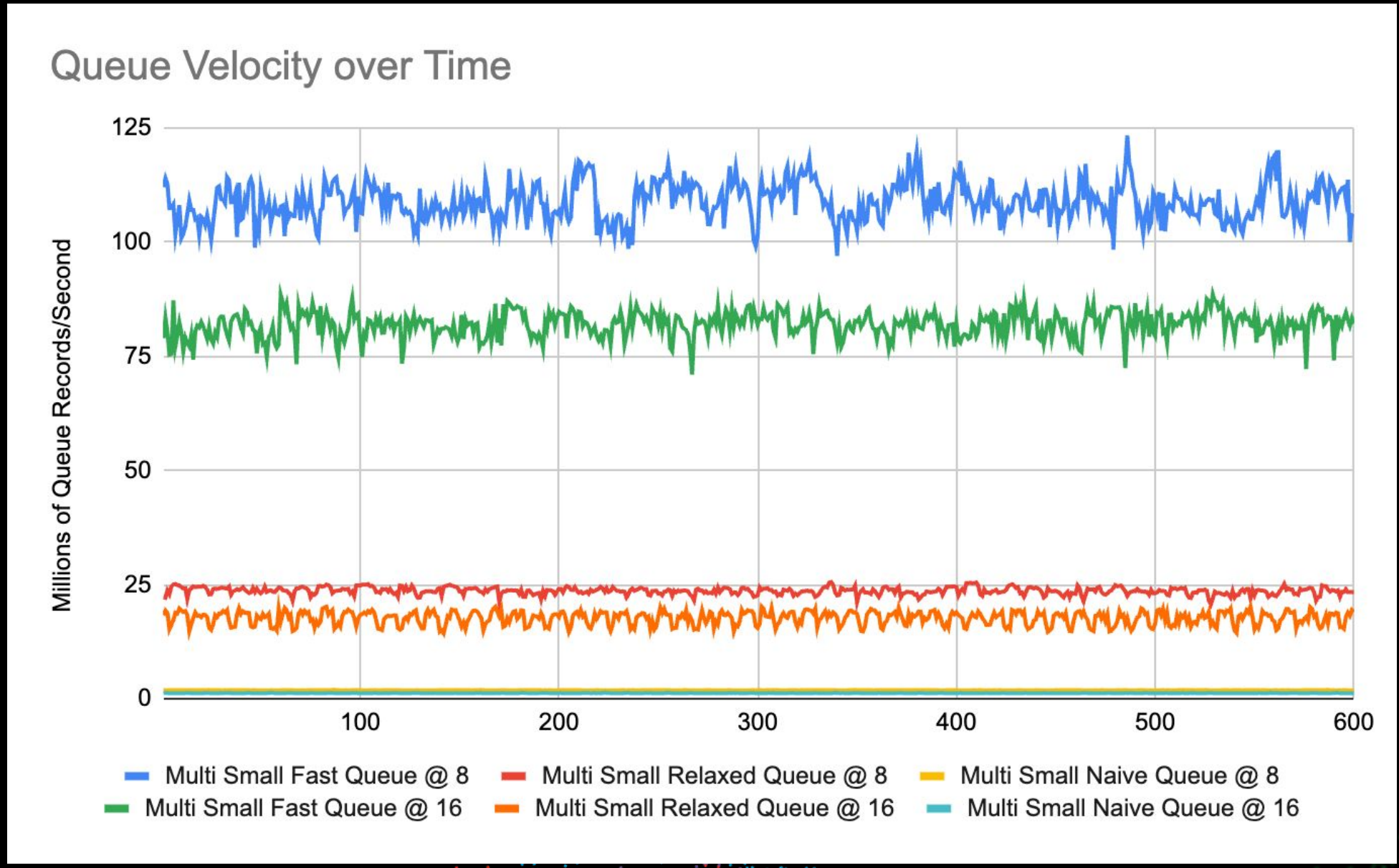
Multi-Consumer Broadcast Queue

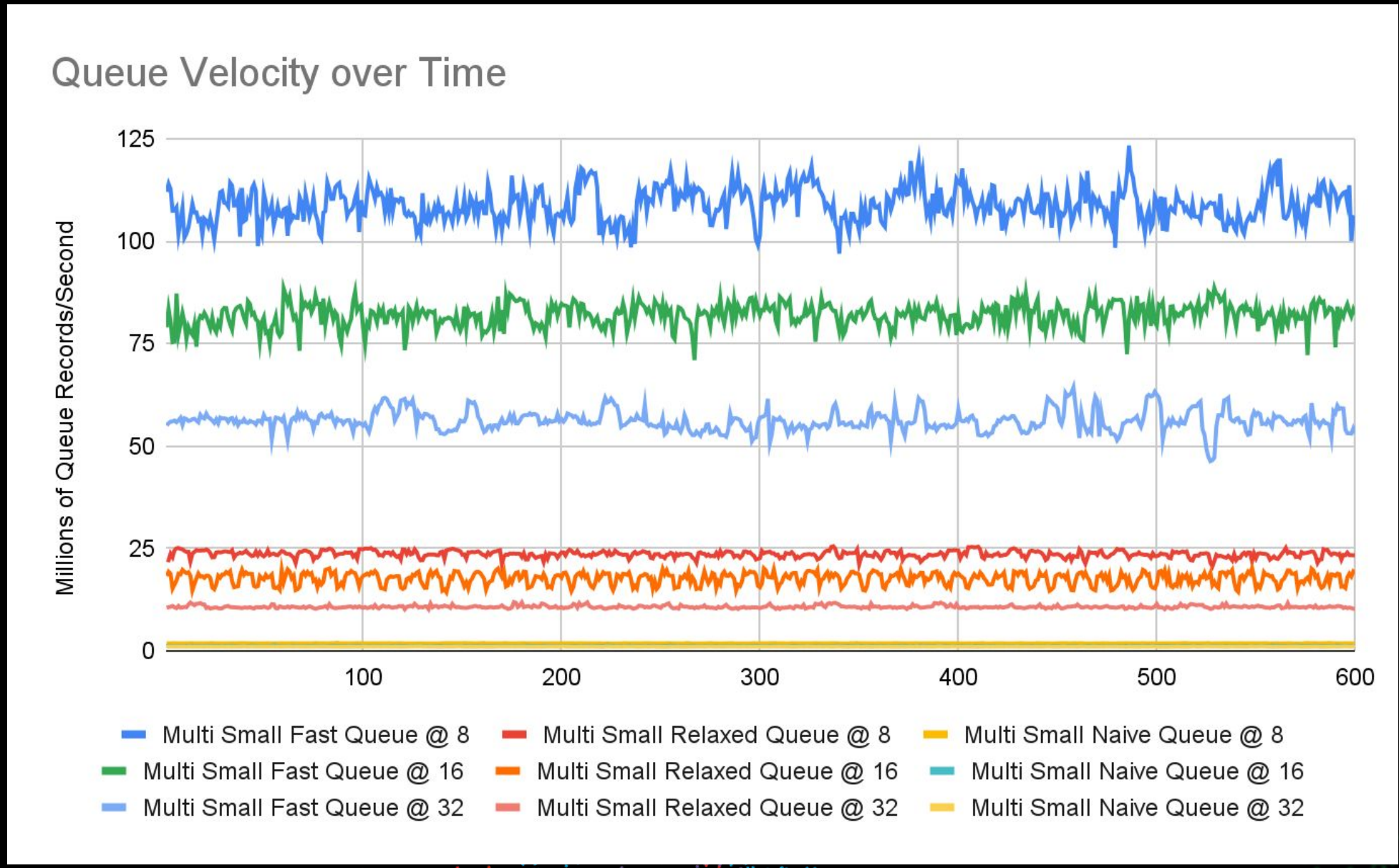
```
struct ring_buffer { // Would be a class template in real code
    struct alignas(cache_line_size) consumer_ctxt {
        consumer_ctxt() : next_consumer_record(0) {}
        std::atomic<std::size_t> next_consumer_record;
    };

    ring_buffer() : next_producer_record(0), slowest_consumer_record(0) {}
    inline void push(int64_t record) noexcept;
    inline int64_t pop(std::size_t id) noexcept;
    inline void collect() noexcept;

    alignas(cache_line_size) std::atomic<std::size_t> next_producer_record;
    alignas(cache_line_size) std::atomic<std::size_t> slowest_consumer_record;
    std::array<consumer_ctxt, num_consumers> consumer_indexes;

    alignas(cache_line_size) std::array<int64_t, capacity> data;
};
```

Multi-Consumer Broadcast Queue

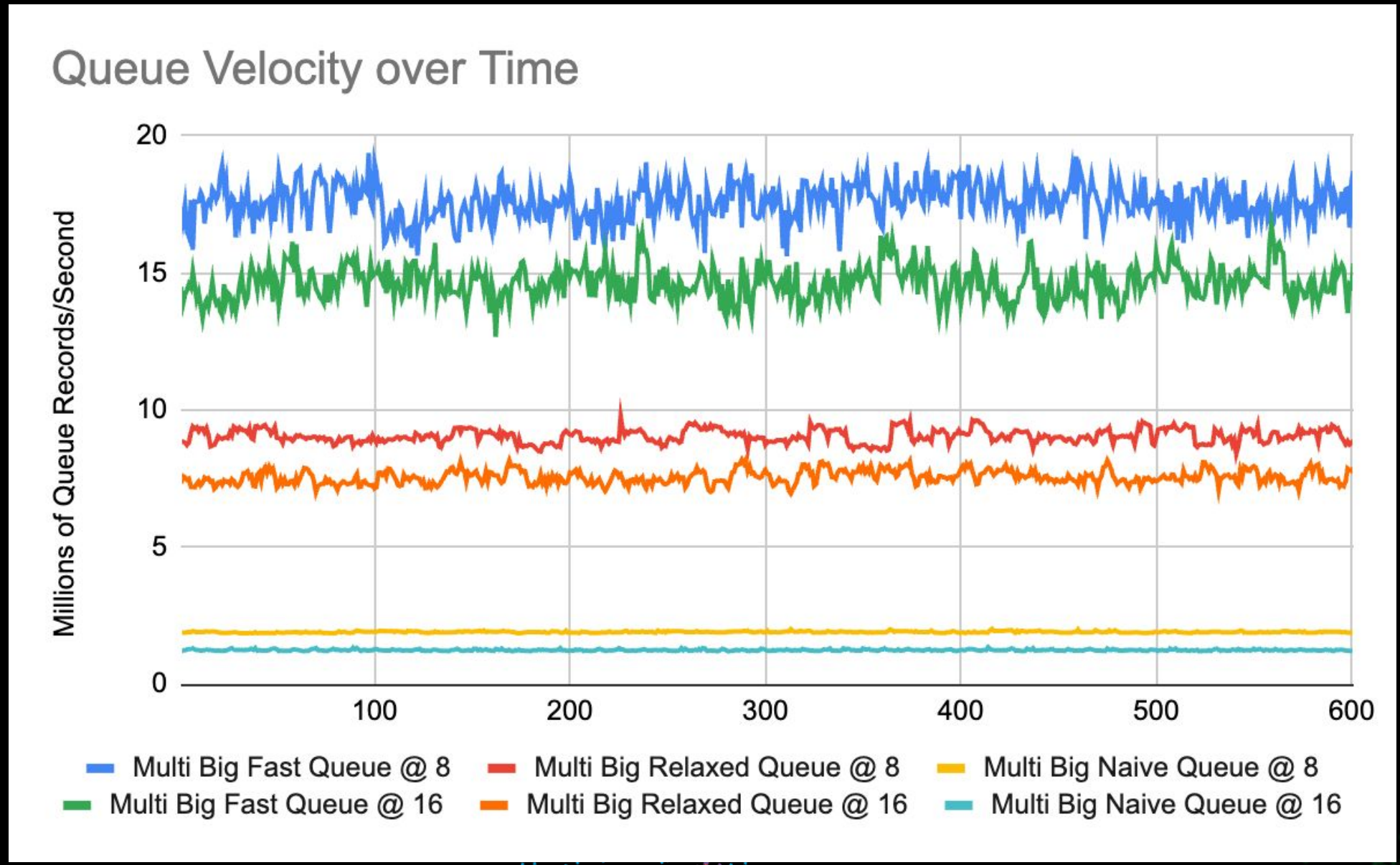
```
inline void ring_buffer::collect() noexcept {
    std::size_t slowest = std::numeric_limits<std::size_t>::max();
    for (auto const& considx : consumer_indexes) {
        if (considx.next_consumer_record < slowest) {
            slowest =
                considx.next_consumer_record.load(std::memory_order::relaxed);
        }
    }
    slowest_consumer_record.store(slowest, std::memory_order::release);
}

void collector(ring_buffer* queue) {
    while (runflag) queue->collect();
}
```


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
100
200
300
400
500
600
Multi Small Fast Queue @ 8
Multi Small Relaxed Queue @ 8
Multi Small Naive Queue @ 8


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
100
200
300
400
500
600
Multi Small Fast Queue @ 8
Multi Small Relaxed Queue @ 8
Multi Small Naive Queue @ 8
Multi Small Fast Queue @ 16
Multi Small Relaxed Queue @ 16
Multi Small Naive Queue @ 16


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
100
200
300
400
500
600
Multi Small Fast Queue @ 8
Multi Small Relaxed Queue @ 8
Multi Small Naive Queue @ 8
Multi Small Fast Queue @ 16
Multi Small Relaxed Queue @ 16
Multi Small Naive Queue @ 16
Multi Small Fast Queue @ 32
Multi Small Relaxed Queue @ 32
Multi Small Naive Queue @ 32


Queue Velocity over Time
Millions of Queue Records/Second
20
15
10
5
0
100
200
300
400
500
600
Multi Big Fast Queue @ 8
Multi Big Relaxed Queue @ 8
Multi Big Naive Queue @ 8


Queue Velocity over Time
Millions of Queue Records/Second
20
15
10
5
0
100
200
300
400
500
600
Multi Big Fast Queue @ 8
Multi Big Relaxed Queue @ 8
Multi Big Naive Queue @ 8
Multi Big Fast Queue @ 16
Multi Big Relaxed Queue @ 16
Multi Big Naive Queue @ 16


Queue Velocity over Time
Millions of Queue Records/Second
20
15
10
5
0
100
200
300
400
500
600
Multi Big Fast Queue @ 8
Multi Big Relaxed Queue @ 8
Multi Big Naive Queue @ 8
Multi Big Fast Queue @ 16
Multi Big Relaxed Queue @ 16
Multi Big Naive Queue @ 16
Multi Big Fast Queue @ 32
Big Relaxed Queue @ 32
Big Naive Queue @ 32

An Interesting Thought

- With the acquire/release optimization, this code now issues zero runtime memory fences; Do we actually need to use `std::atomic` here?
- What does it give us?
- What happens if we drop it?

Undefined Behavior Queue 😉

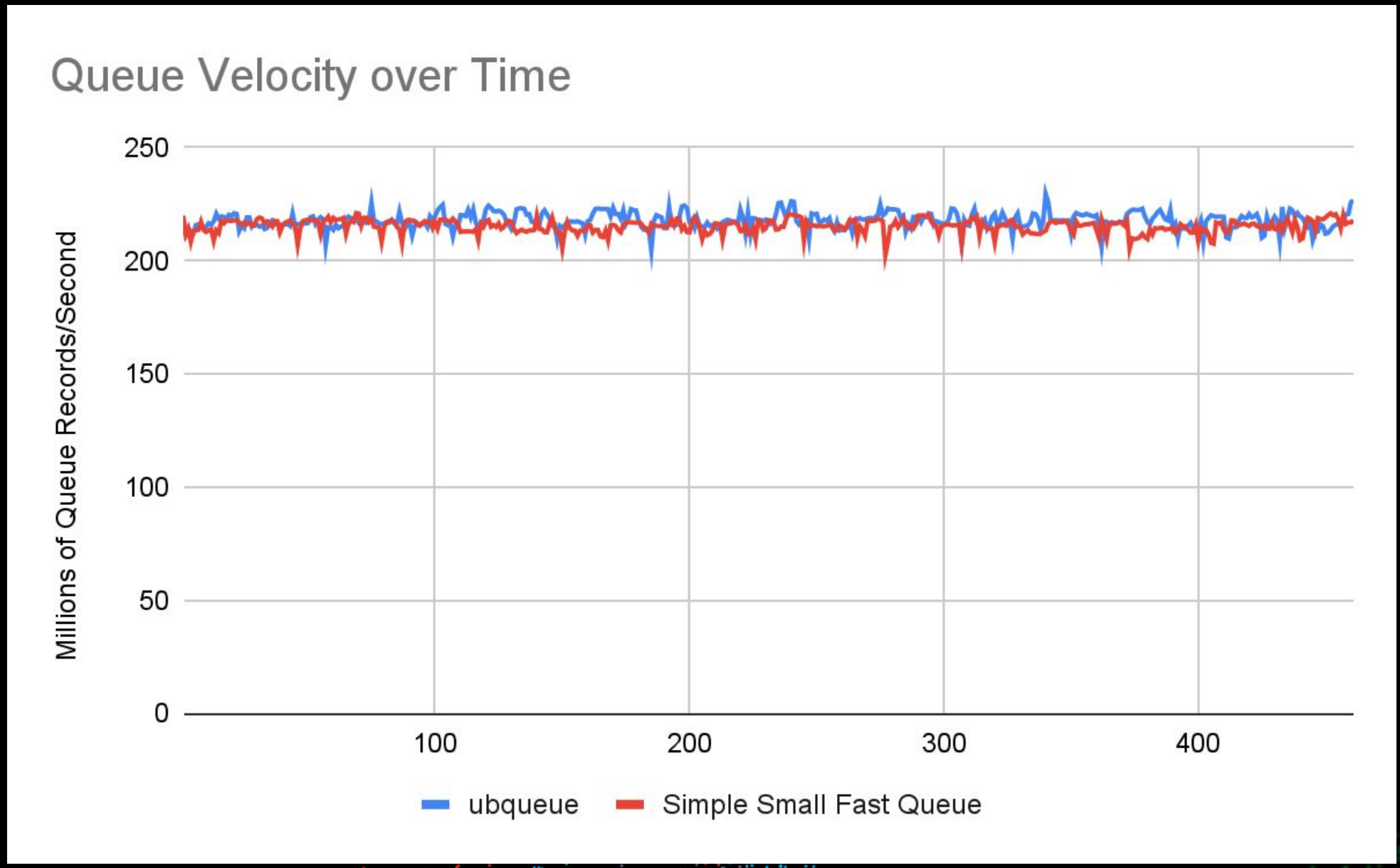
```
// Bad code! Do not use!
struct ubqueue { // Would be a class template in real code, but this is bad code anyways :)
    static constexpr std::size_t capacity = 32'768;
    static constexpr std::size_t cache_line_size = std::hardware_destructive_interference_size;

    ubqueue() : next_producer_record(0), next_consumer_record(0) {}

    inline void push(int64_t record) noexcept;

    inline int64_t pop() noexcept;

    /* details */

    alignas(cache_line_size) std::size_t volatile next_producer_record;
    alignas(cache_line_size) std::size_t volatile next_consumer_record;
    alignas(cache_line_size) std::array<int64_t, num_records> data;
};
```

Undefined Behavior Queue 😉

```
inline void ubqueue::push(int64_t record) noexcept  {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer = next_producer_record;
    while (!has_space(producer)) sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    std::atomic_signal_fence(std::memory_order::acq_rel); // load/store compiler fence
    next_producer_record = producer + 1;
}
```

Undefined Behavior Queue 😉

```
inline int64_t ubqueue::pop() noexcept {
    // Wait until there's data
    std::size_t spin_counter = 0;
    std::size_t consumer = next_consumer_record;
    while (!has_data(consumer)) sleep(spin_counter);

    // Consume the record
    auto record = data[map_index(consumer)];
    std::atomic_signal_fence(std::memory_order::acq_rel); // load/store compiler fence
    next_consumer_record = consumer + 1;
    return record;
}
```


Queue Velocity over Time
Millions of Queue Records/Second
250
200
150
100
50
0
100
200
300
400
ubqueue
Simple Small Fast Queue

An Interesting Thought

- 😲 It actually does work, and the performance is essentially identical.
- Comparison of the assembly yields essentially the same codegen.
- Have we shown that `std::atomic` is unnecessary?
- What happens if we try to run the same code on this laptop? 🤔

An Interesting Thought

```
cfretz@XF3QVTQ7K6 talk % build/ubqueue
Queue is moving at 0 records/second.
Assertion failed: (val == counter), function consumer, file ubqueue.cc, line 93.
zsh: abort      build/ubqueue
```

An Interesting Thought

- Good to know our assertions work 🤣
- What went wrong? Why does this work on x86_64, but not on ARM64?

Platform Guarantees

- From the Intel x86_64 architecture specification:
  - Neither Loads Nor Stores Are Reordered with Like Operations
  - Stores Are Not Reordered With Earlier Loads
  - Loads May Be Reordered with Earlier Stores to Different Locations
  - Stores Are Transitively Visible
  - Stores Are Seen in a Consistent Order by Other Processors

Platform Guarantees

- On x86_64, acquire/release semantics *are essentially free*
- This is *not* true on more weakly-ordered platforms like ARM and PowerPC
- ARM requires explicit load-acquire and store-release opcodes to accomplish the same

Atomic Optimization

```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer =
        next_producer_record.load(
            std::memory_order::acquire);
    while (!has_space(producer))
        sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    next_producer_record.store(
        producer + 1, std::memory_order::release);
}
```

```
.LBB0_1:
        and     x8, x23, #0x7fff        ;; map queue index
        str     x20, [x21, x8, lsl #3] ;; store rec in queue
        add     x20, x20, #1            ;; incr curr index
        add     x8, x23, #1             ;; incr prod index
        cmp     x20, x22                ;; completion check
        stlr    x8, [x19]               ;; store prod index
        b.eq    .LBB0_10                ;; producer terminates
.LBB0_2:
        add     x8, x19, #64
        ldar    x23, [x19]              ;; load prod index
        ldar    x8, [x8]                ;; load consumer index
        sub     x8, x23, x8             ;; get records in queue
        cmp     x8, #8, lsl #12         ;; queue capacity check
        b.lo    .LBB0_1                 ;; space is available
        mov     x24, xzr
        b       .LBB0_5                 ;; nanosleep backoff
```

Platform Specific Failure

- What causes the failure on ARM? What are we missing?
- Fence instructions are missing, but why do they matter?
- Why do we *actually* need the fences?

Platform Specific Failure

- Does it matter if the producer reads the consumers' operations out of order?
- Does it matter if the producer reads the collector's operations out of order?
- Does it matter if the consumers read the producer's operations out of order?

Atomic Optimization

```
inline void
ring_buffer::push(int64_t record) noexcept {
    // Wait until there's space
    std::size_t spin_counter = 0;
    std::size_t producer =
        next_producer_record.load(
            std::memory_order::acquire);
    while (!has_space(producer))
        sleep(spin_counter);

    // Produce the record
    data[map_index(producer)] = record;
    next_producer_record.store(
        producer + 1, std::memory_order::release);
}
```

```
.LBB0_1:
        and     x8, x23, #0x7fff        ;; map queue index
        str     x20, [x21, x8, lsl #3] ;; store rec in queue
        add     x20, x20, #1            ;; incr curr index
        add     x8, x23, #1             ;; incr prod index
        cmp     x20, x22                ;; completion check
        stlr    x8, [x19]               ;; store prod index
        b.eq    .LBB0_10                ;; producer terminates
.LBB0_2:
        add     x8, x19, #64
        ldar    x23, [x19]              ;; load prod index
        ldar    x8, [x8]                ;; load consumer index
        sub     x8, x23, x8             ;; get records in queue
        cmp     x8, #8, lsl #12         ;; queue capacity check
        b.lo    .LBB0_1                 ;; space is available
        mov     x24, xzr
        b       .LBB0_5                 ;; nanosleep backoff
```

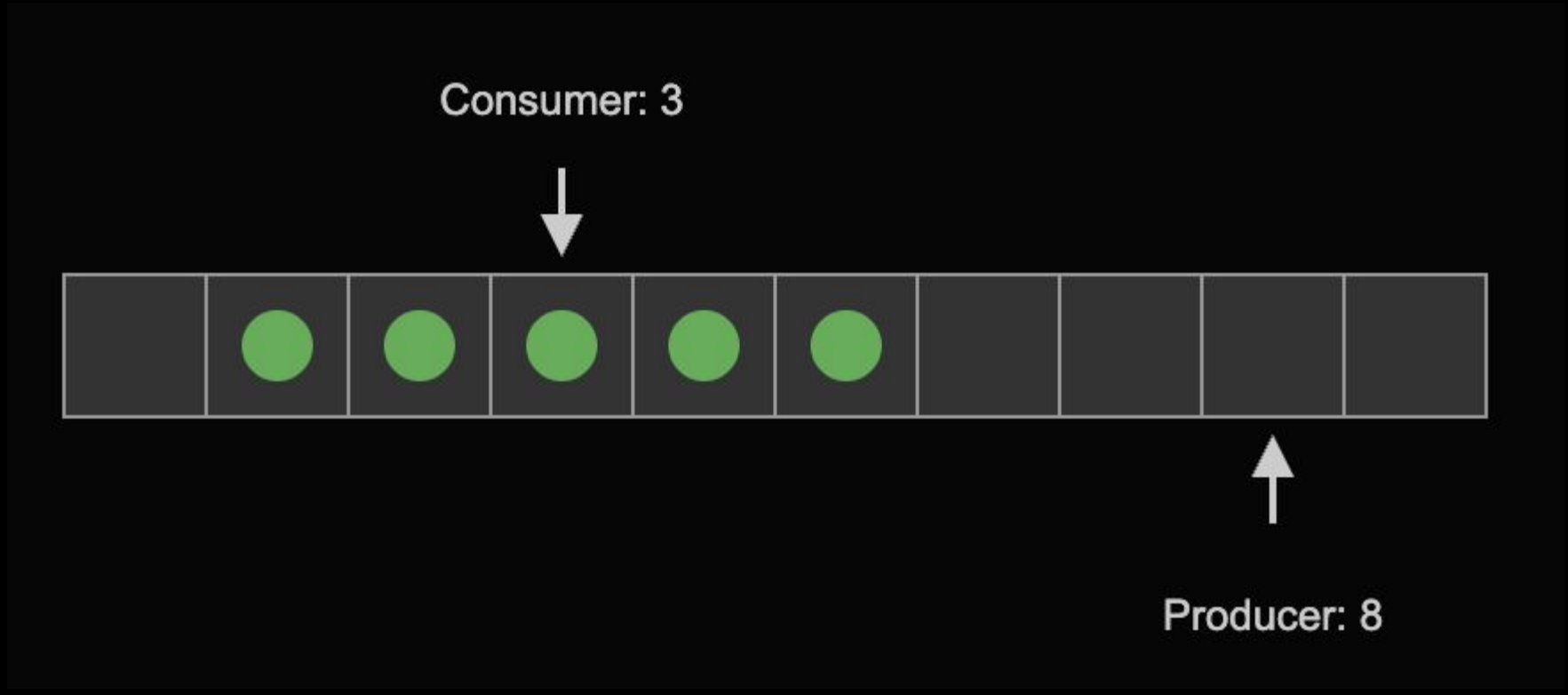
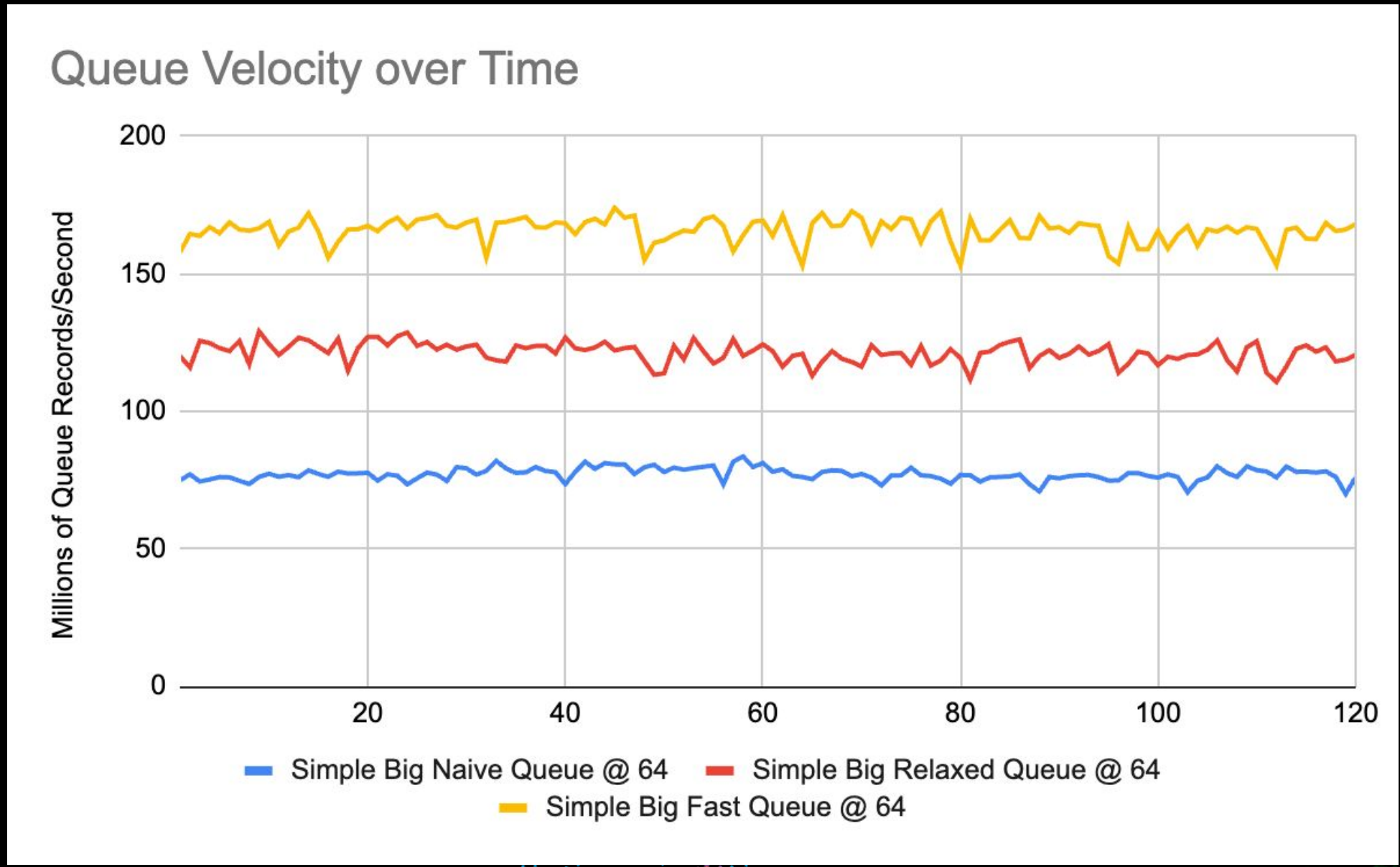


Image generated with ChatGPT 4o Image Gen

Platform Specific Failure

- Without the release fence on the producer side, and the acquire fence on the consumer side, the consumer can see the producer move forward before its queue updates are visible.
- Consumer then reads junk from the previous iteration.
- `std::atomic` ensures we always get the correct behavior on all platforms.


Queue Velocity over Time
Millions of Queue Records/Second
200
150
100
50
0
20
40
60
80
100
120
Simple Big Naive Queue @ 64
Simple Big Relaxed Queue @ 64
Simple Big Fast Queue @ 64

The Final Optimization

- Further research encountered this paper published in 2009.
- Suggests *further* optimizing cache performance with the introduction of "cached" indexes for both the producer and consumer.
- Adds an additional, cache line aligned, index for both the producer and consumer that holds a previous snapshot of the opposite index.
- Cached copies are updated *only* when the producer/consumer would otherwise need to sleep (cached index appears full/empty).
- Resolves the final remaining cause of cache misses for the simple queue.

The Final Optimization

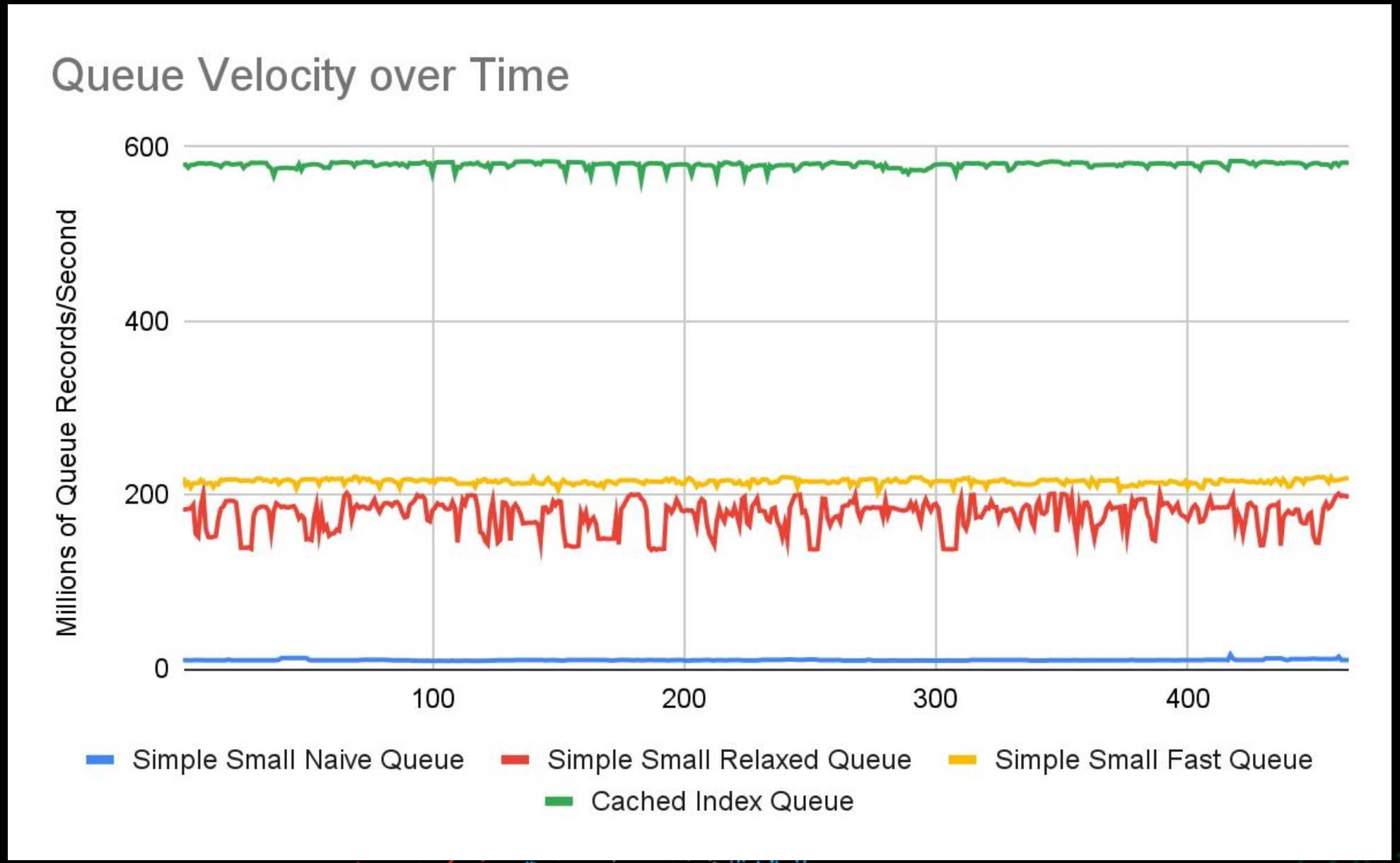
```cpp
struct ring_buffer { // Would be a class template in real code
    static constexpr std::size_t capacity = 32'768;
    static constexpr std::size_t cache_line_size = std::hardware_destructive_interference_size;

    ring_buffer() : next_producer_record(0), next_consumer_record(0) {}
    inline void push(int64_t record) noexcept;
    inline int64_t pop() noexcept;

    /* details */

    alignas(cache_line_size) std::atomic<std::size_t> next_producer_record;    // real producer index
    alignas(cache_line_size) std::atomic<std::size_t> producer_consumer_cache; // cache of consumer for producer

    alignas(cache_line_size) std::atomic<std::size_t> next_consumer_record;    // real consumer index
    alignas(cache_line_size) std::atomic<std::size_t> consumer_producer_cache; // cache of producer for consumer

    alignas(cache_line_size) std::array<int64_t, capacity> data;
};
```

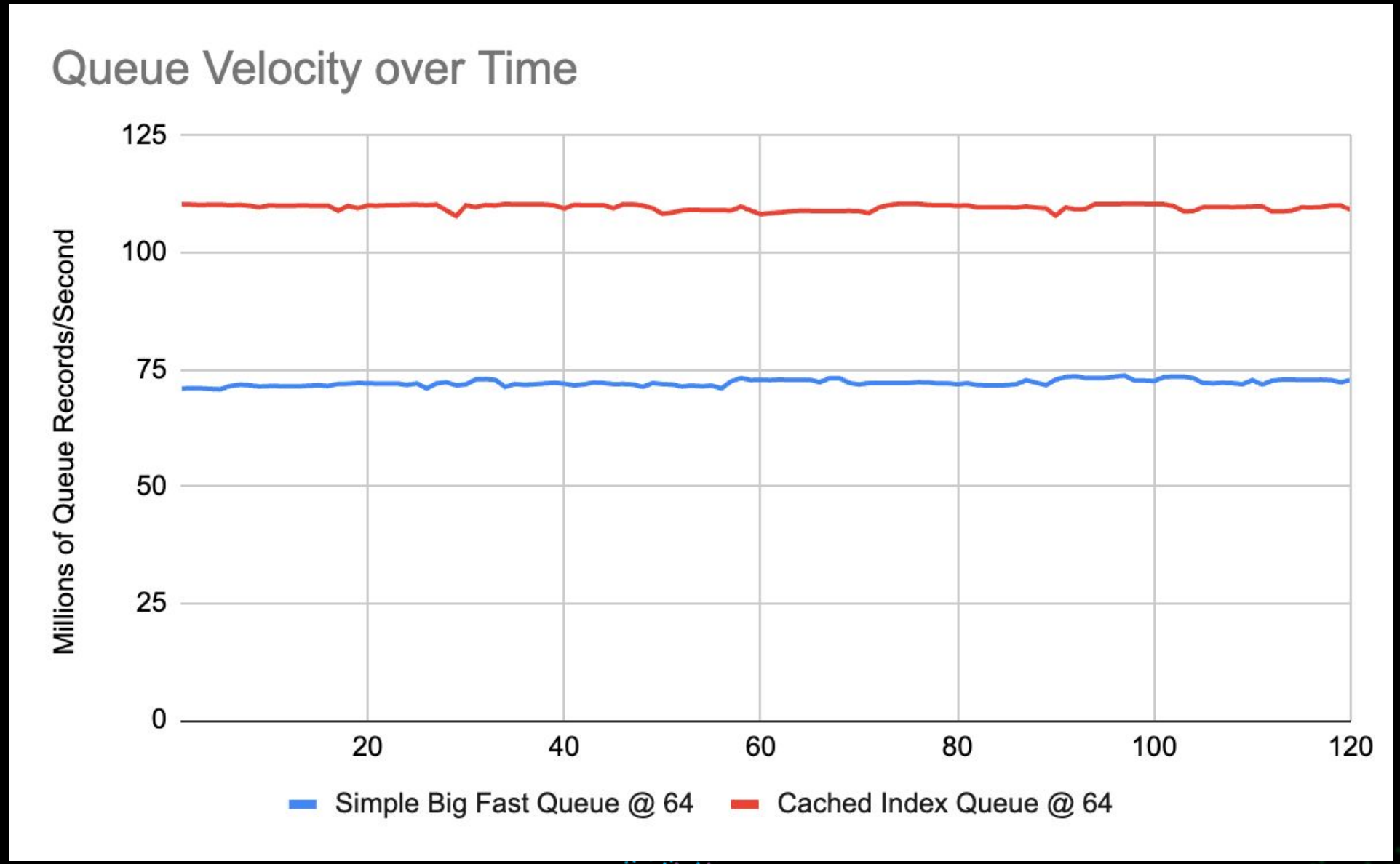
The Final Optimization

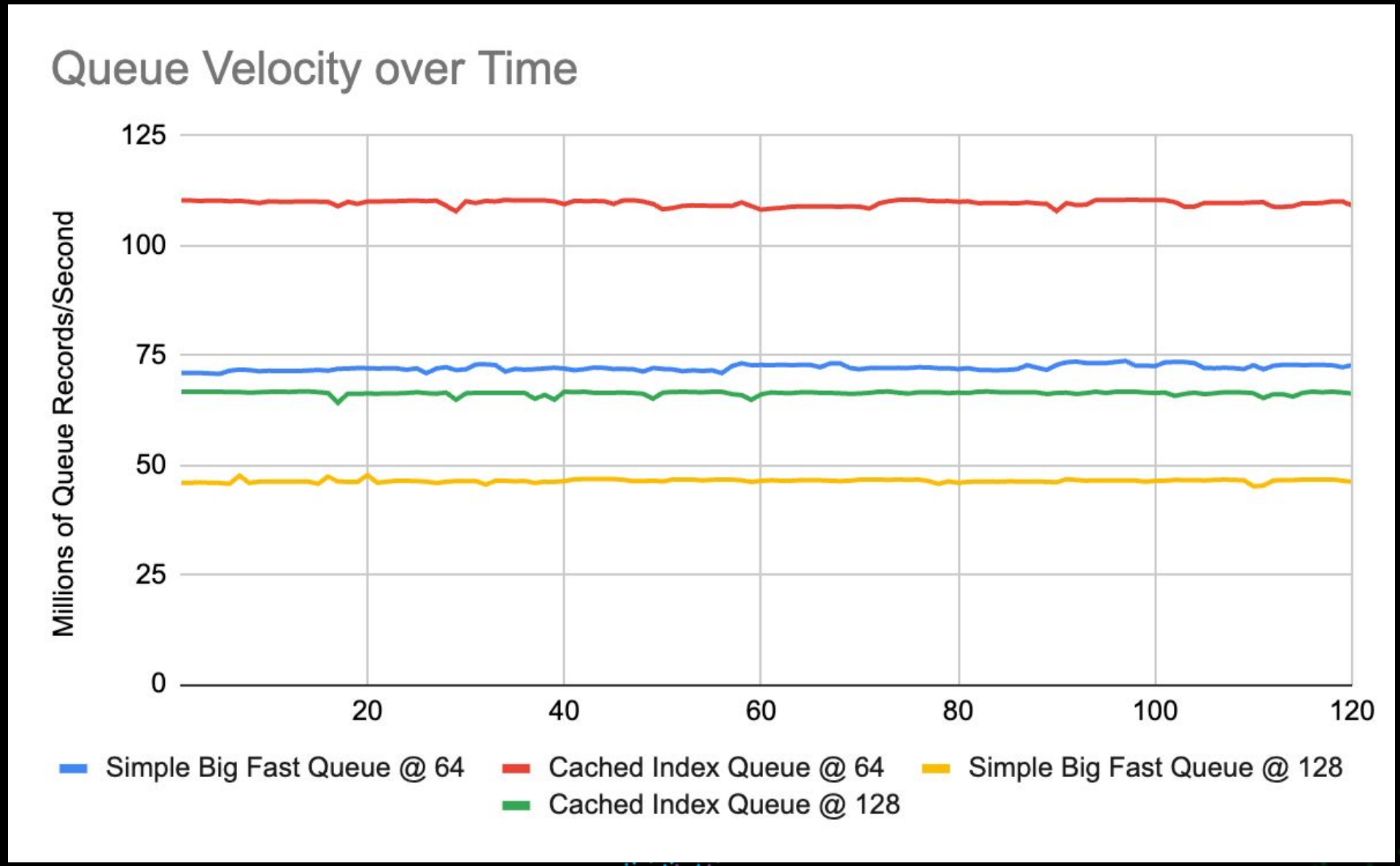
```
 Performance counter stats for 'build/simplesmallfastqueue':

      29735927127      L1-dcache-loads:u
       1798844339      L1-dcache-load-misses:u   #    6.05% of all L1-dcache accesses
        664376080      cache-misses:u            #   32.238 % of all cache refs
       2060853784      cache-references:u

     30.300179622 seconds time elapsed

     34.977956000 seconds user
      0.683142000 seconds sys
```

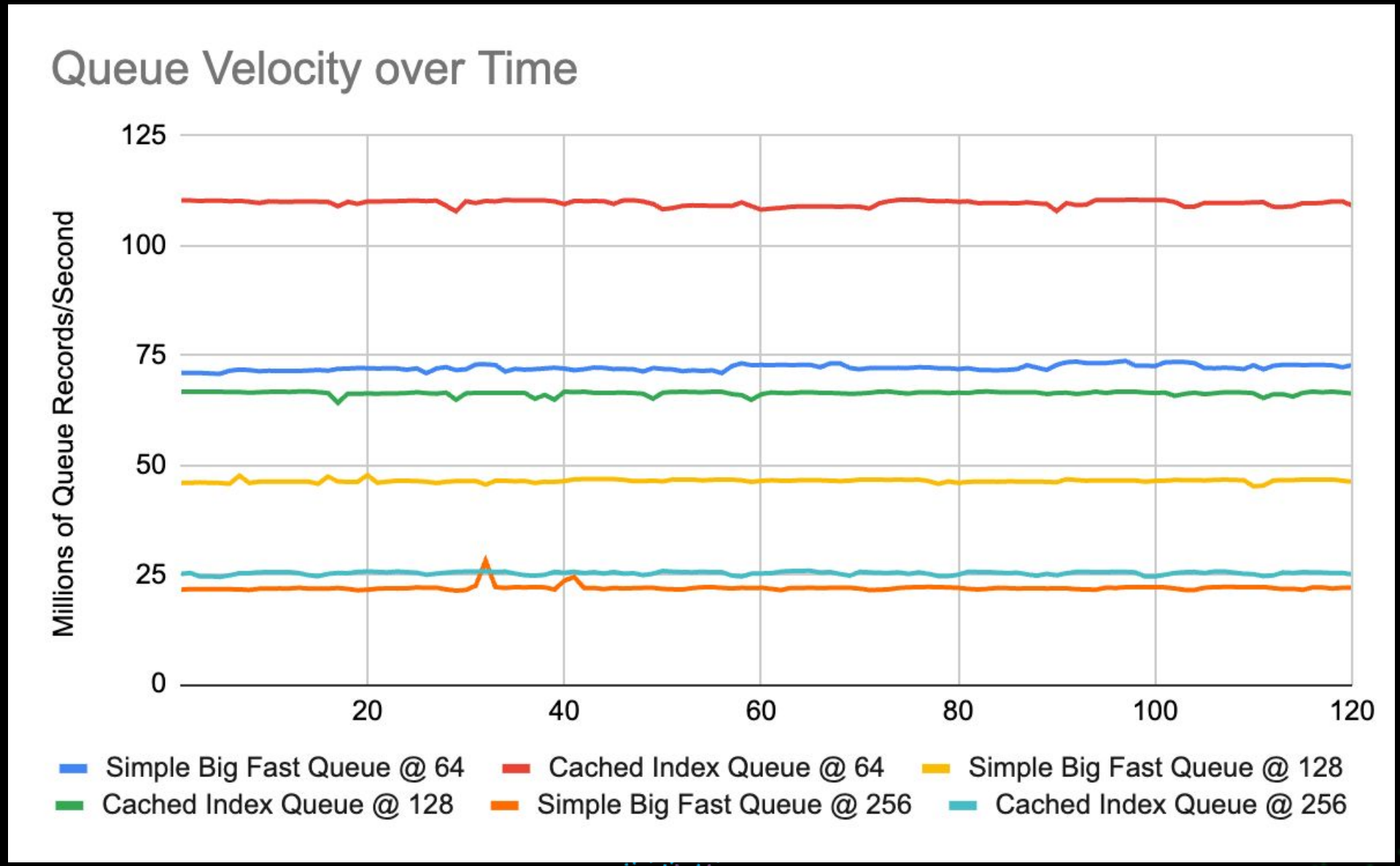
The Final Optimization

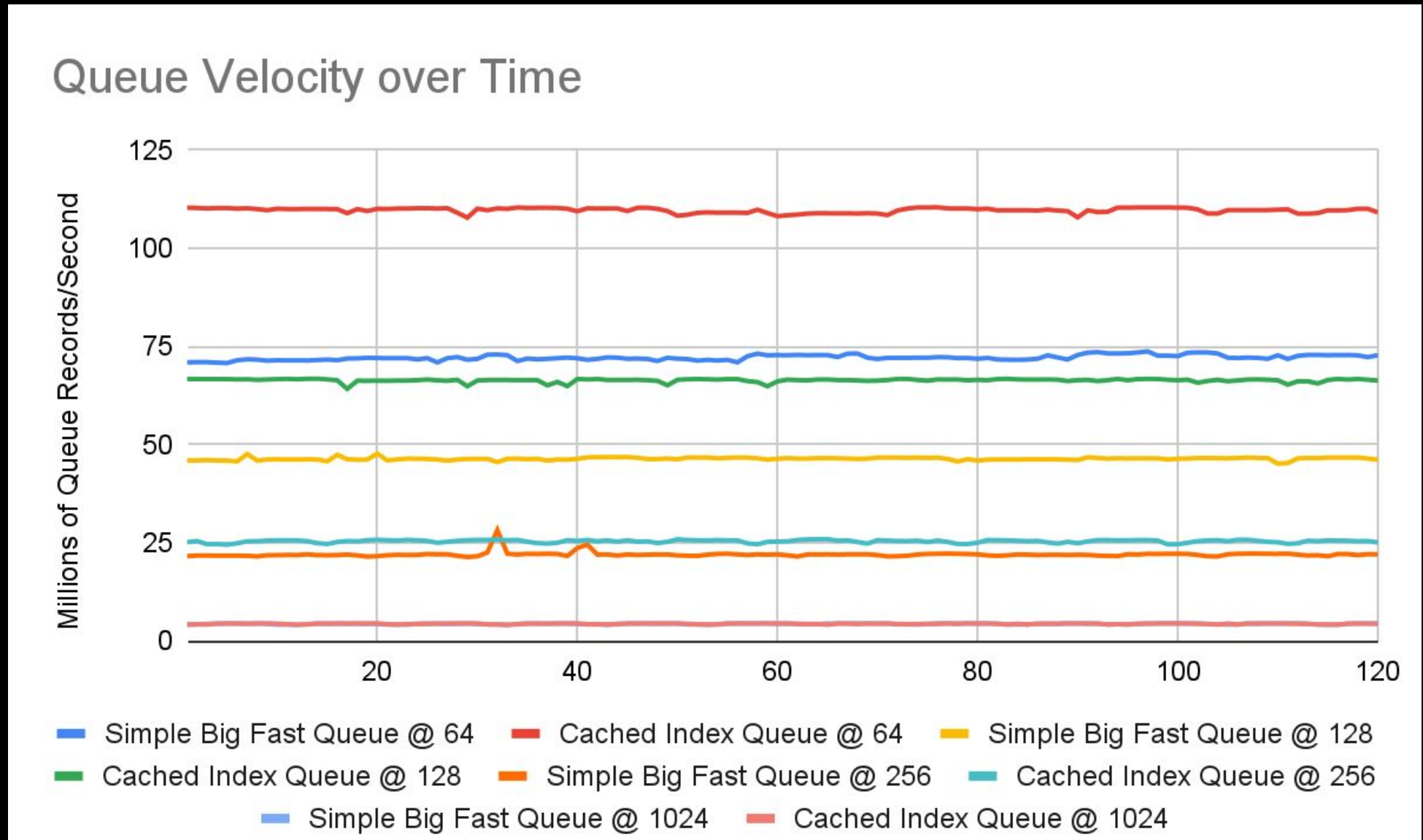
```
 Performance counter stats for 'build/smalloptimqueue':

      87822085858      L1-dcache-loads:u
       4410973235      L1-dcache-load-misses:u   #    5.02% of all L1-dcache accesses
            54422      cache-misses:u            #    0.001 % of all cache refs
       4431111302      cache-references:u

     30.306062641 seconds time elapsed

     56.443693000 seconds user
      0.125849000 seconds sys
```


Queue Velocity over Time
600
400
200
0
Millions of Queue Records/Second
100
200
300
400
Simple Small Naive Queue
Simple Small Relaxed Queue
Simple Small Fast Queue
Cached Index Queue


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
20
40
60
80
100
120
Simple Big Fast Queue @ 64
Cached Index Queue @ 64


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
20
40
60
80
100
120
Simple Big Fast Queue @ 64
Cached Index Queue @ 64
Simple Big Fast Queue @ 128
Cached Index Queue @ 128


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
20
40
60
80
100
120
Simple Big Fast Queue @ 64
Cached Index Queue @ 64
Simple Big Fast Queue @ 128
Cached Index Queue @ 128
Simple Big Fast Queue @ 256
Cached Index Queue @ 256


Queue Velocity over Time
Millions of Queue Records/Second
125
100
75
50
25
0
20
40
60
80
100
120
Simple Big Fast Queue @ 64
Cached Index Queue @ 64
Simple Big Fast Queue @ 128
Cached Index Queue @ 128
Simple Big Fast Queue @ 256
Cached Index Queue @ 256
Simple Big Fast Queue @ 1024
Cached Index Queue @ 1024

The Final Optimization

- 🤯
- For small queue sizes specifically, optimization makes a *massive* difference.
- Our queue passes integers, a real world use-case would be pointers.
- Decays as data sizes get large.
- Never seems to perform noticeably worse.
- Optimization doesn’t appear to be implemented by *either* folly or boost.

Final Caveats

- This is a data structure that optimizes particularly well.
- Performance diff in this case is *massive* because the data structure is so naturally well aligned with acquire/release semantics.
- Often not possible to remove all read-modify-writes.
- fetch_add and CAS on x86 *always* requires a full fence (`lock` prefix), regardless of what you ask for.
- Benchmark!

Surprising Results

- Cache line sizes on my M3 Macbook are 128 bytes long
- Optimum memory alignment in my testing was 64 bytes; No idea why
- AppleClang does not have `std::hardware_destructive_interference_size`

Conclusions

- Using the right model for the job can make a *massive* difference.
- Performance can be hard to reason about; sometimes, it yields surprises.
- The standard memory models do an *excellent* job of ensuring you “can’t write better yourself.”
- Relaxed atomics can be difficult to reason about, but can be worth it in the right context.

Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers

Bloomberg

Engineering

TechAtBloomberg.com