

# C++ ONLINE

DAVE ROWLAND

LOCK-FREE QUEUES IN THE  
MULTIVERSE OF MADNESS

2026

# Lock-free Queues in the Multiverse of Madness

Dave Rowland - C++ Online 12/03/26





# Meeting C++ 2019

**David Rowland  
Fabian Renn-Giles**

**Real-time 101**

## Real-time 101

David Rowland & Fabian Renn-Giles

*@drowaudio @hogliux*

**ADC 19**

Real-time 101 - part I: Investigating the real-time problem space

Fabian Renn-Giles & Dave Rowland

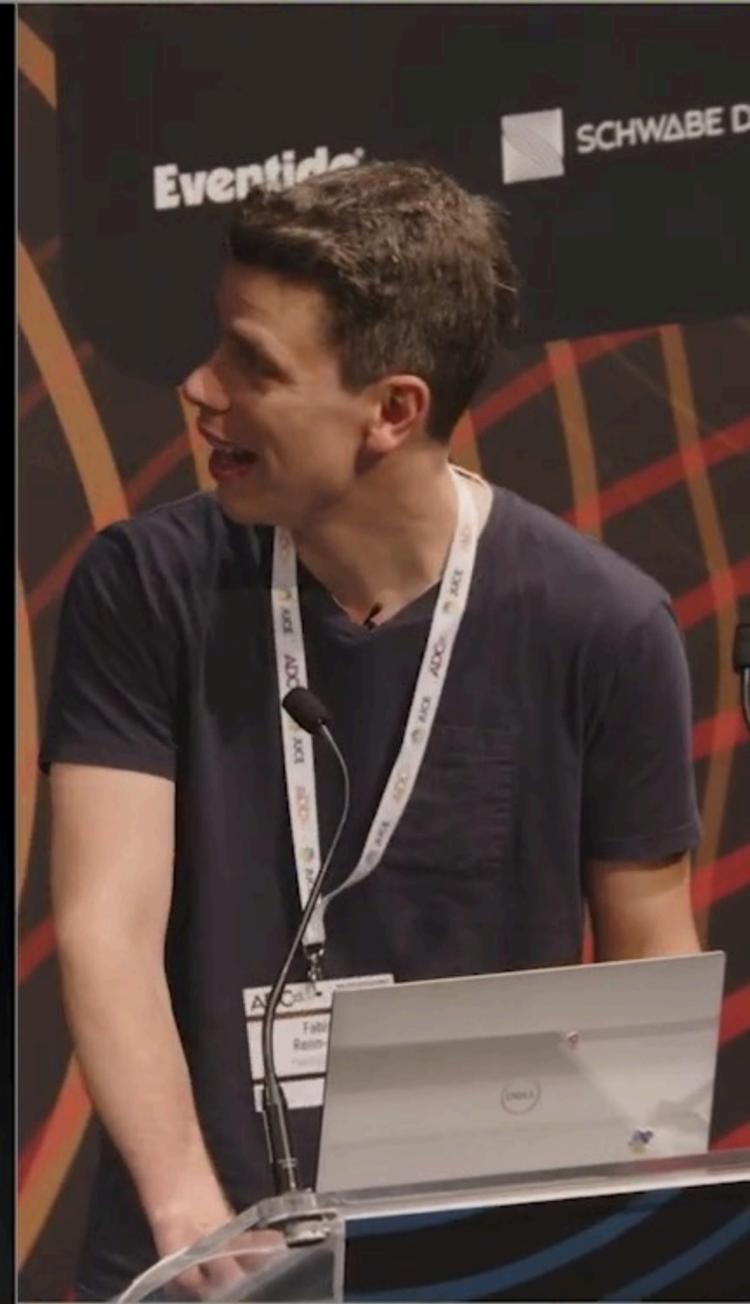
**ADC 19**

Real-time 101 - Part II: The real-time audio developer's toolbox

Dave Rowland & Fabian Renn-Giles



# Sins & Confessions



**FABIAN RENN-GILES**

# Blocking vs. Non-wait-free vs. Wait Free

**Blocking**

**Non-wait-free**

**Wait-free**

May context switch for example due to a lock, system call etc.

Execution time is unbounded

Execution time is bounded\*

Caches likely to be invalidated

Must contain a loop (which is unbounded)

No unbounded loops

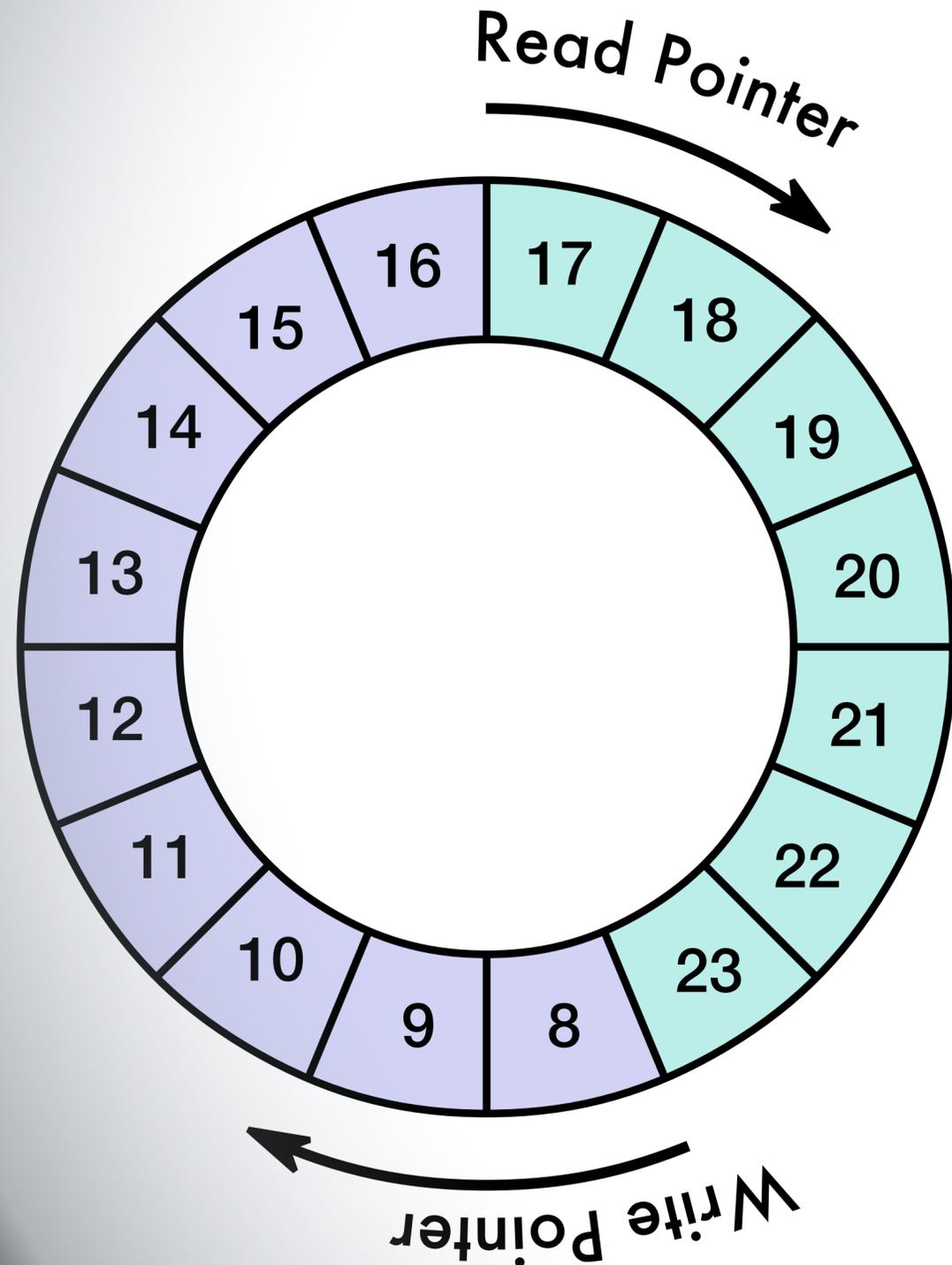
Memory may be swapped

Blocking operations are never wait-free (but not vice versa)

**David Rowland  
Fabian Renn-Giles**

**Real-time 101**

# The humble FIFO



- Realtime code use ring buffers to implement a FIFO
- Fixed capacity: no allocations (i.e. realtime safe)
- Various flavours

# The humble FIFO

Single Consumer, Single Producer

```
template <typename T> class fifo {
public:
    bool push (T && arg) {
        auto pos = writepos.load();
        auto next = (pos + 1) % slots.size();

        if (next == readpos.load())
            return false;

        slots[pos] = std::move (arg);
        writepos.store (next);
        return true;
    }

    bool pop(T& result) {
        auto pos = readpos.load();

        if (pos == writepos.load())
            return false;

        result = std::move (slots[pos]);
        readpos.store ((pos + 1) % slots.size());
        return true;
    }
private:
    std::vector<T> slots = {}; std::atomic<int> readpos = {0}, writepos = {0};
};
```

David Rowland  
Fabian Renn-Giles

Real-time 101

# Costs of various FIFOs

		Single Producer		Multiple Producer	
		Report Full	Overwrite on Full	Report Full	Overwrite on Full
Single Consumer	Report Empty				
	"null" on Empty				
Multiple Consumer	Report Empty				
	"null" on Empty				

Wait free on read

Wait free on write

Wait free on read and write

Not wait free on write or read

 boost::lockfree::spscqueue

 boost::lockfree::queue

**ER** rigtorp::SPSCQueue

**ER** rigtorp::MPMCQueue

 juce::AbstractFifo

 folly::MPMCQueue

 choc::SingleReaderSingleWriterFIFO

 choc::MultipleReaderMultipleWriterFIFO

 tbb::concurrent\_bounded\_queue

**ES** es::lockfree::mpmc\_queue

 moodycamel::ReaderWriterQueue

 moodycamel::ConcurrentQueue

 farbot::fifo<spsc>

 farbot::fifo<spmcs>

 farbot::fifo<mpsc>

 farbot::fifo<mpmcs>

More to the story?



What is a Queue?



**GRAND OPENING**  
— CELEBRATION —



**GRAND OPENING**  
— CELEBRATION —

# What is a Queue?

FIFO not LIFO (stack)

- Transferring data between threads
  - Audio samples (input or for recording)
  - MIDI messages
  - Metering levels
  - Co-operative processing thread pools
  - Network data
- Filter coefficients
- Playback graphs
- Non-audio
  - Trading data
  - Kernel data
  - Task queues

# The Problem with Locks

- System calls
  - Unbounded - not real-time safe
  - Verify with RTSan
- Spin locks (user-space)
  - Thread starvation
  - Priority inversion
  - Don't scale (thread contention)

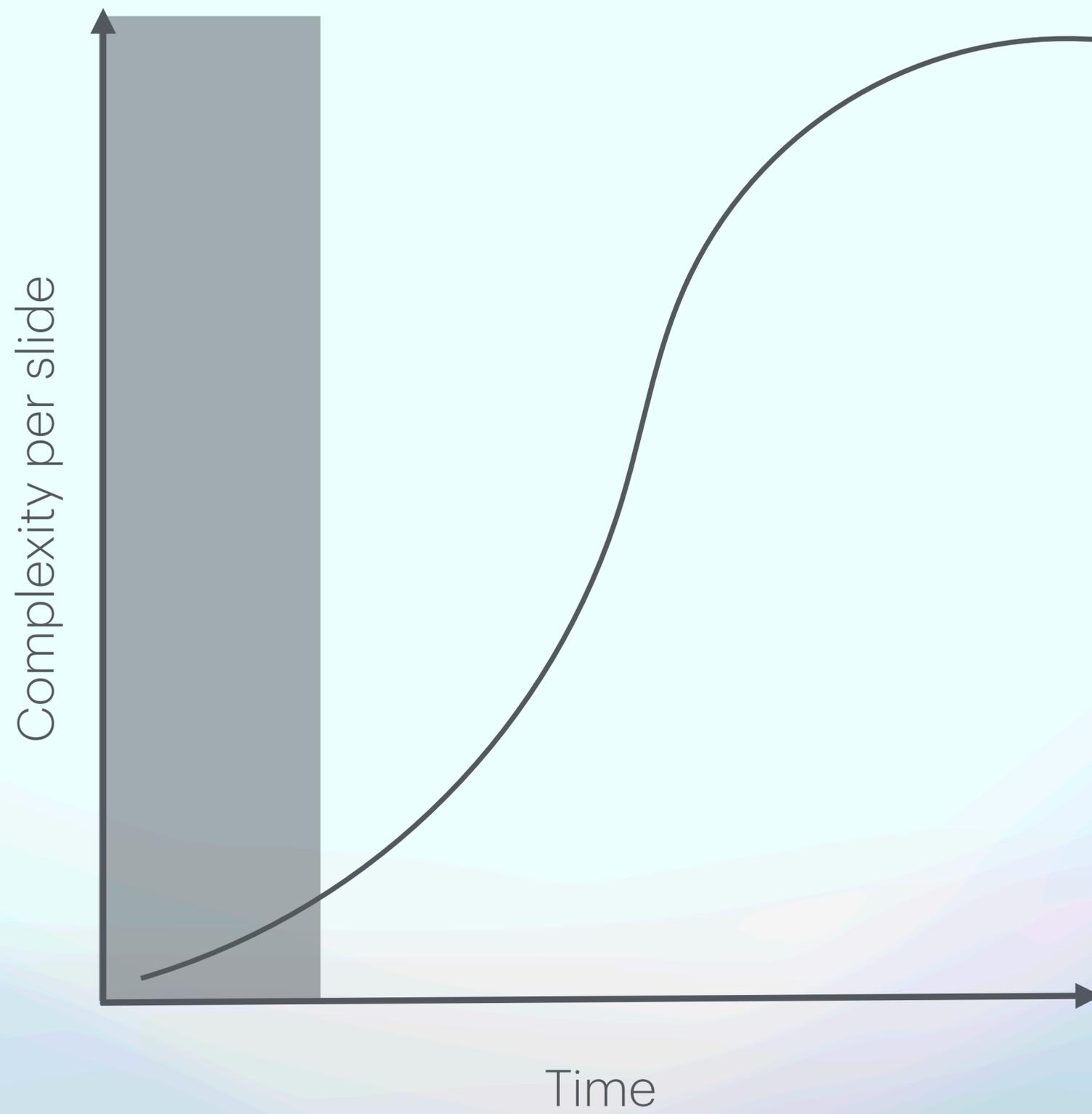
# Simple Behaviour

## SPSC

<b>push</b>	Block on full*	Overwrite when full	Fail/try_
<b>pop</b>	Block on empty*	Return default on empty	Fail/try_
<b>capacity</b>	Static (compile-time)	Fixed (runtime)	Dynamic (grows)
<b>Bulk push/pop</b>	No - single item	Yes - multiple items	
<b>Message size</b>	Fixed	Dynamic	
<b>Message size limit</b>	Limited (8 bytes?)	Unlimited	
<b>Triviality</b>	Trivial	Non-trivial	
<b>Gurantees</b>	None - blocking	Lock-free	Wait-free
<b>Num processes</b>	Single	Inter-process	



```
main.cpp
mpmc
  boost_lockfree_queue.h
  choc_mpmc.h
  farbot_mpmc_wf.h
  farbot_mpmc.h
  lockfree_queue_mpmc_v5.h
  lockfree_queue_mpmc_v6.h
  lockfree_queue_mpmc_v9.h
  moodycamel_concurrentqueue.h
  mutex_queue.h
  realtime_mutex_queue.h
  rigtorp_mpmc.h
spsc
  farbot_spsc_wf.h
  farbot_spsc.h
  lockfree_queue_spsc_v5_aba_bug.h
  lockfree_queue_spsc_v5.h
  lockfree_queue_spsc_v6.h
  lockfree_queue_spsc_v7.h
  lockfree_queue_spsc_v8.h
  lockfree_queue_spsc_v9.h
spsc
  boost_lockfree_spscqueue.h
  choc_spsc.h
  farbot_spscqueue.h
  lockfree_queue_v1.h
  lockfree_queue_v2.h
  lockfree_queue_v3.h
  lockfree_queue_v4.h
  lockfree_queue_v5.h
  lockfree_queue_v6.h
  lockfree_queue_v7_1.h
  lockfree_queue_v7_2.h
  lockfree_queue_v7.h
  lockfree_queue_v8.h
  lockfree_queue_v9.h
  moodycamel_concurrentqueue_spsc.h
  moodycamel_readerwriterqueue.h
  rigtorp_spscqueue.h
  tbb_concurrent_queue.h
st
  utils.h
```



# Benchmarks

- Microbenchmarks can be misleading...
- Idealised to show impact of queue
- Queues just store **ints**
- In practice, what else happens in the system has a big impact on performance (cache thrashing etc.)

# Benchmark Caveats

- All benchmarks are within a single process
- On an MacBook Pro M2 Max (with 8 cores, the number of P-cores)

```
template<typename queue_type>
void measure_serial(benchmark::State& state)
{
    const size_t queue_size = state.range(0);
    const size_t iters = 4 * queue_size;

    for (auto _ : state)
    {
        // Initialise the queue half full
        queue_type queue(queue_size);

        for (size_t i = 0; i < (queue_size / 2); ++i)
            push (queue, i);

        stopwatch sw;

        for (size_t i = 0; i < iters; ++i)
        {
            if (! queue.try_push (i))
                continue;

            int v;
            if (! queue.try_pop (v))
                continue;

            benchmark::DoNotOptimize(v);
        }

        state.SetIterationTime(sw.get());
    }

    // Throughput is calculated as an item being enqueued and dequeued, so travelling fully through the queue
    state.SetItemsProcessed(int64_t(state.iterations()) * iters);
}
```

```

template<typename T>
class queue
{
public:
    queue (size_t capacity_)
        : capacity (capacity_)
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    size_t head { 0 }, tail { 0 };

    size_t next_index (size_t current) const
    {
        return (current + 1) % capacity;
    }
};

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}

```

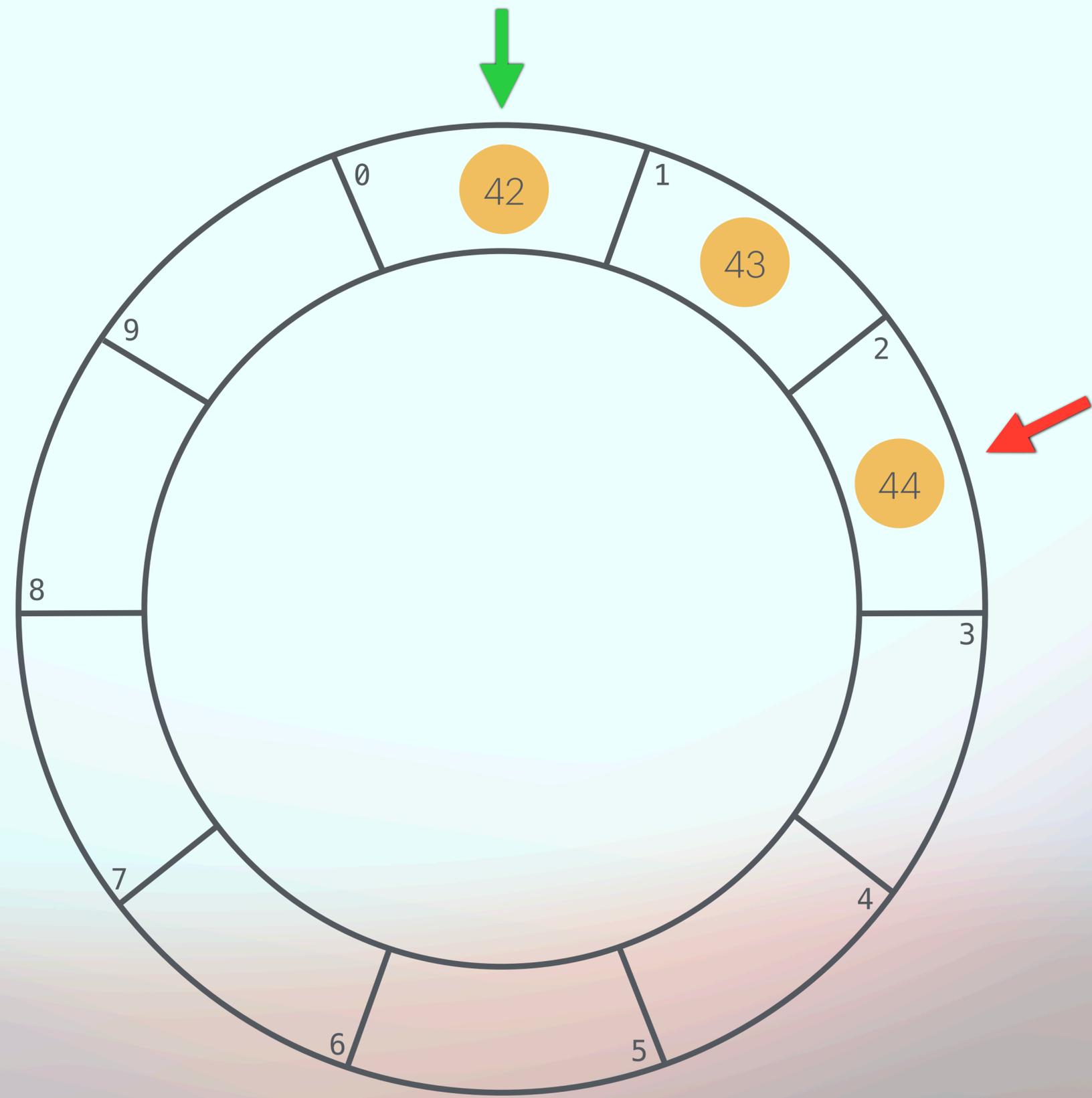
```

bool try_pop (T& v)
{
    size_t current_head = head;

    if (current_head == tail)
        return false;

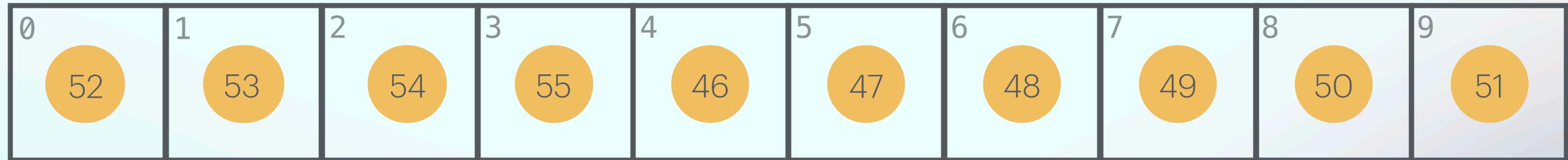
    v = data[current_head];
    head = next_index (current_head);
    return true;
}

```



tail == head -> **EMPTY**

tail/  
write

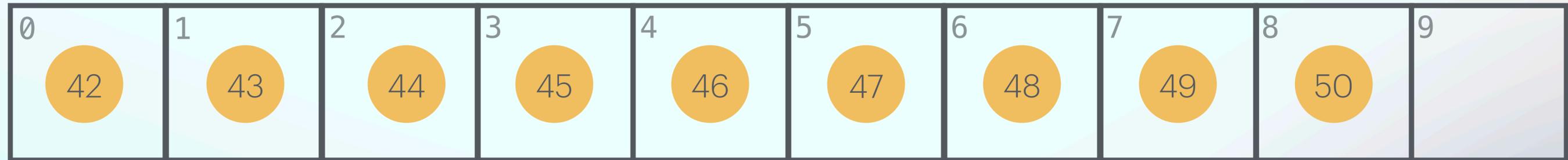


head/  
read



`(next)tail == head -> FULL`

tail/  
write/  
push



head/  
read/  
pop



**capacity** = size - 1

182,987,000

2.00E+08

1.50E+08

1.00E+08

5.00E+07

0.00E+00

items/s

queue



```

template<typename T>
class queue
{
public:
    queue (size_t capacity_)
    : capacity (capacity_)
    {
    }

    bool try_push (const T&);
    bool try_pop (T&);

private:
    
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    size_t head { 0 }, tail { 0 };

    size_t next_index (size_t current) const {
        return (current + 1) % capacity;
    }
};

```

```

bool try_push (const T& v)
{
    

    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}

```

```

bool try_pop (T& v)
{
    

    size_t current_head = head;

    if (current_head == tail)
        return false;

    v = data[current_head];
    head = next_index (current_head);
    return true;
}

```

```

template<queue_end queue_end, typename queue_type>
struct queue_thread
{
    queue_thread(queue_type &queue_, std::latch &start_latch_, size_t num_iterations)
        : queue(queue_), start_latch(start_latch_), iters(num_iterations)
    {}

    void run_async()
    {
        thread = std::thread([this]
        {
            start_latch.arrive_and_wait();

            stopwatch sw;

            if constexpr (queue_end == queue_end::producer)
            {
                //...
            }
            else
            {
                //...
            }

            res.duration = sw.get();

            run_result = res;
        });
    }

    queue_result join()
    {
        assert (thread.joinable());
        thread.join();
        return run_result;
    }

private:
    queue_type& queue;
    std::latch& start_latch;
    const size_t iters;
    std::thread thread;
    queue_result run_result;
};

```

```

if constexpr (queue_end == queue_end::producer)
{
    for (size_t i = 0; i < iters; ++i)
    {
        if (queue.try_push(i))
        {
        }
    }
}

```

```

else
{
    for (size_t i = 0; i < iters; ++i)
    {
        int v;
        if (queue.try_pop(v))
        {
        }
    }
}

```

**182,987,000**

**41,400,000**

**x4.5**

2.00E+08

1.50E+08

1.00E+08

5.00E+07

0.00E+00

**queue**

**mutex\_queue**



```

class realtime_mutex
{
    std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
    // Non-blocking try_lock for real-time contexts
    bool try_lock() noexcept
    {
        // First test: cheap read-only check
        if (flag.test (std::memory_order_relaxed))
            return false; // Already locked

        // Then test-and-set if the first test passed
        return ! flag.test_and_set (std::memory_order_acquire);
    }

    // Blocking lock with test, test-and-set optimization
    void lock() noexcept
    {
        while (true)
        {
            // Optimistically assume the lock is free on the first try
            if (! flag.test_and_set (std::memory_order_acquire))
                break; // Successfully acquired the lock

            // Wait for lock to be released without generating cache misses
            while (flag.test (std::memory_order_relaxed))
                CPU_PAUSE(); // CPU-specific pause instruction
        }
    }

    void unlock() noexcept
    {
        flag.clear (std::memory_order_release);
    }
};

```

<https://rigtorp.se/spinlock/>

## Lightning Talk: A Spinlock Implementation

- Fedor Pikus
- CppNow 2022

The screenshot shows a presentation slide for a lightning talk. On the left, there is a video of Fedor Pikus speaking. The main content is a code editor displaying the spinlock implementation. The code includes headers for `<atomic>` and `<time>`, and defines a `spinlock` class with `lock()` and `unlock()` methods. The `lock()` method uses a loop with `flag.test_and_set` and `CPU_PAUSE()` to wait for the lock. The `unlock()` method uses `flag.store` to release the lock. The slide also features logos for C++ Now 2022, Sonar, and Jet Brains, and the text 'A Spinlock Implementation' and 'C++ Now.org'.

182,987,000

x4.5

41,400,000

20,178,300

x0.5

2.00E+08

1.50E+08

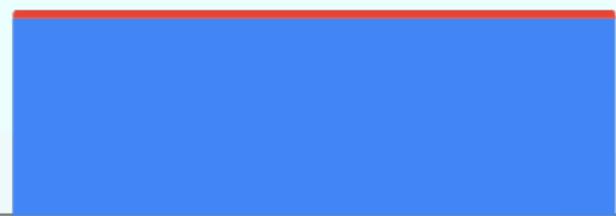
1.00E+08

5.00E+07

0.00E+00



queue



mutex\_queue



realtime\_mutex\_queue

```

template<typename T>
class drow_queue_v1
{
public:
    drow_queue_v1 (size_t capacity_)
        : capacity (capacity_)
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    std::atomic<size_t> head { 0 }, tail { 0 };

    size_t next_index (size_t current) const
    {
        return (current + 1) % capacity;
    }
};

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}

```

```

bool try_pop (T& v)
{
    size_t current_head = head;

    if (current_head == tail)
        return false;

    v = data[current_head];
    head = next_index (current_head);
    return true;
}

```

5.00E+08

4.00E+08

3.00E+08

2.00E+08

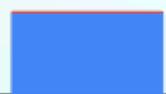
1.00E+08

0.00E+00

**mutex\_queue**

**realtime\_mutex\_queue**

**drow\_spsc\_v1**



```

template<typename T>
class drow_queue_v1
{
public:
    drow_queue_v1 (size_t capacity_)
        : capacity (capacity_)
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    std::atomic<size_t> head { 0 }, tail { 0 };

    size_t next_index (size_t current) const
    {
        return (current + 1) % capacity;
    }
};

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}

```

```

bool try_pop (T& v)
{
    size_t current_head = head;

    if (current_head == tail)
        return false;

    v = data[current_head];
    head = next_index (current_head);
    return true;
}

```

$2^3 \text{ capacity} = 8 = 1000b$

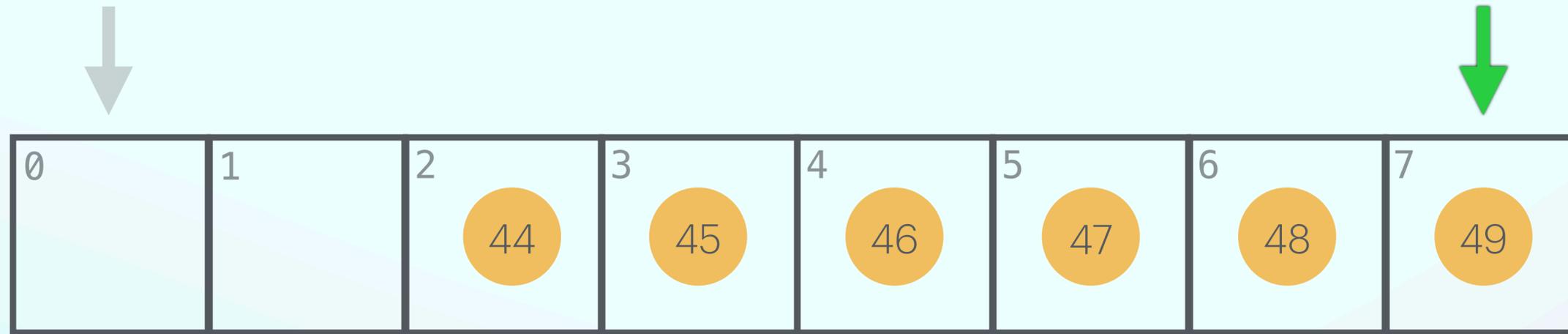
$\text{capacity} - 1 = 7 = 0111b$

$\text{next\_tail} = 7 + 1 = 8 = 1000b$

$\& 0111b$

$= 0000b$

tail/  
write



head/  
read

$2^2 \text{ capacity} = 4 = 100b$

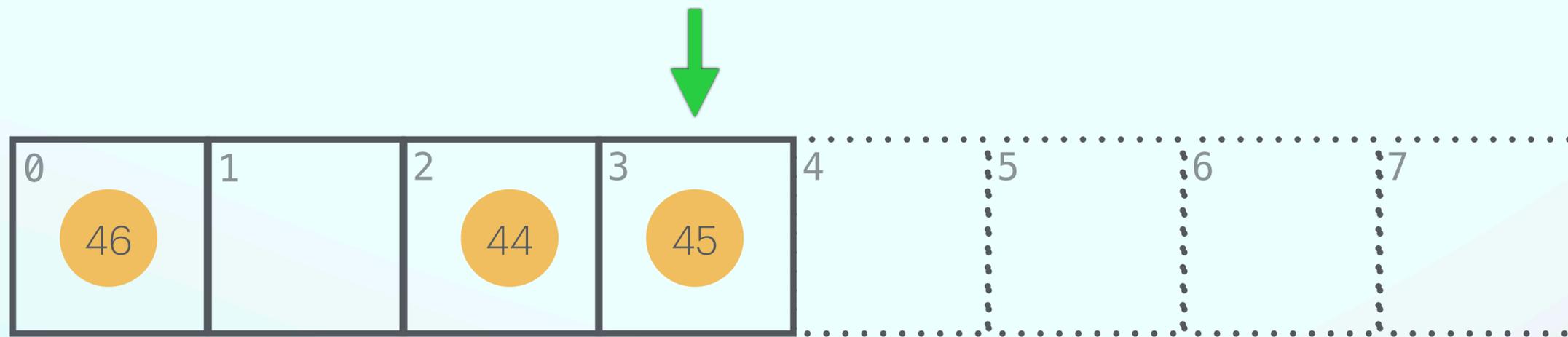
$\text{capacity} - 1 = 3 = 011b$

$++\text{tail} = 4$

$\text{index} = \text{tail} \& 011b$

$= 0000b$

tail/  
write



head/  
read



2 1 0 = EMPTY

$\text{size} = \text{tail} - \text{head}$

$5 - 2 = 3$

= FULL

# Optimisations

## Monotonic head/tail counters

- Will head/tail wrap around?
  - $2^{64} = 18,446,744,073,709,551,616 = 1.845 \times 10^{19}$
  - `queue` throughput was **182,987,000** ( $1.82 \times 10^8$ ) items/second
  - $1.845 \times 10^{19} / 1.82 \times 10^8$   
= **100,809,041,482s**  
= **3,197** years

```

template<typename T>
class drow_queue_v2
{
public:
    drow_queue v2 (size_t capacity )
        : capacity (std::bit_ceil (capacity_))
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    std::atomic<size_t> head { 0 }, tail { 0 };
};

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail.load();
    size_t current_head = head.load();

    size_t size = current_tail - current_head;

    if (size >= (capacity - 1)) // full
        return false;

    size_t index = current_tail & (capacity - 1);
    data[index] = v;
    tail.store (current_tail + 1);

    return true;
}

```

```

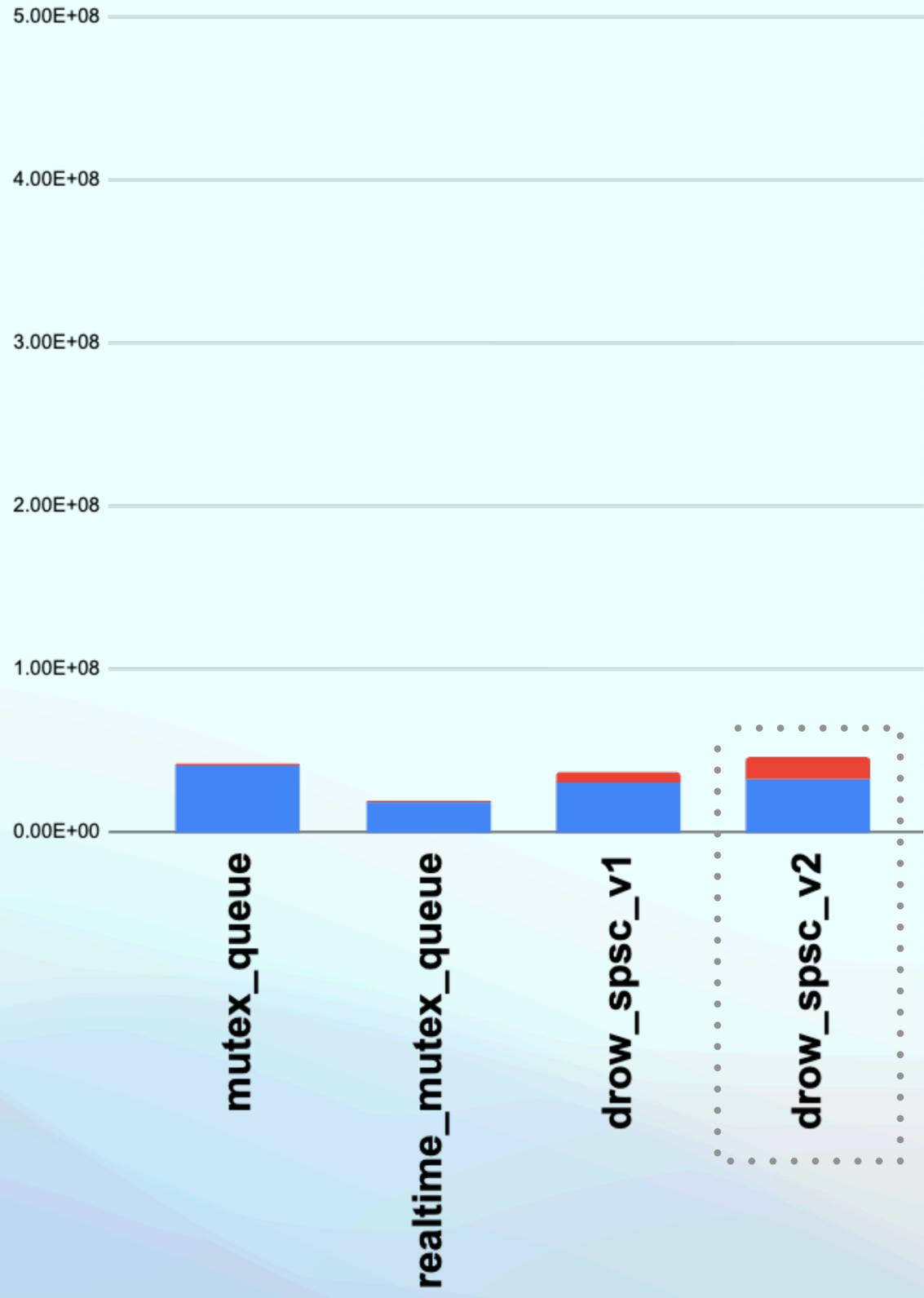
bool try_pop (T& v)
{
    size_t current_head = head.load();
    size_t current_tail = tail.load();

    if (current_head == current_tail) // empty
        return false;

    size_t index = current_head & (capacity - 1);
    v = data[index];
    head.store(current_head + 1);

    return true;
}

```



# Optimisations

## Memory Ordering

- See:
  - Timur's ADC 25 talk: "Demystifying std::memory\_order"
  - Herb Sutter: "atomic Weapons"
- Ordering can happen at:
  - Compiler/hoisting (code)
  - Optimiser (instruction)
  - CPU (microcode)

The C++ memory model defines the rules for how multiple threads can access shared memory, specifying when writes by one thread become visible to other threads and what synchronisation is needed to avoid data races.

# Memory Ordering

- Relaxed:
  - Only guarantees atomicity
  - Ordering between the same *atomic* variable
- Acquire/release
  - Synchronise with each other
  - Work in pairs (store = release, load = acquire)
  - Allows some reordering
  - Writes before a release store become visible after an acquire load in another thread
- Sequential consistency
  - Acquire/release semantics
  - All threads agree on ordering (consistent with program order)
- Implemented like read/write locks per-cache-line

```

template<typename T>
class drow_queue_v3
{
public:
    drow_queue_v3 (size_t capacity_)
        : capacity (capacity_)
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    std::atomic<size_t> head { 0 }, tail { 0 };

    size_t next_index (size_t current) const
    {
        return (current + 1) % capacity;
    }
};

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail.load(std::memory_order_relaxed);
    size_t next_tail = next_index (current_tail);

    if (next_tail == head.load (std::memory_order_acquire))
        return false;

    data[current_tail] = v;
    tail.store (next_tail, std::memory_order_release);
    return true;
}

```

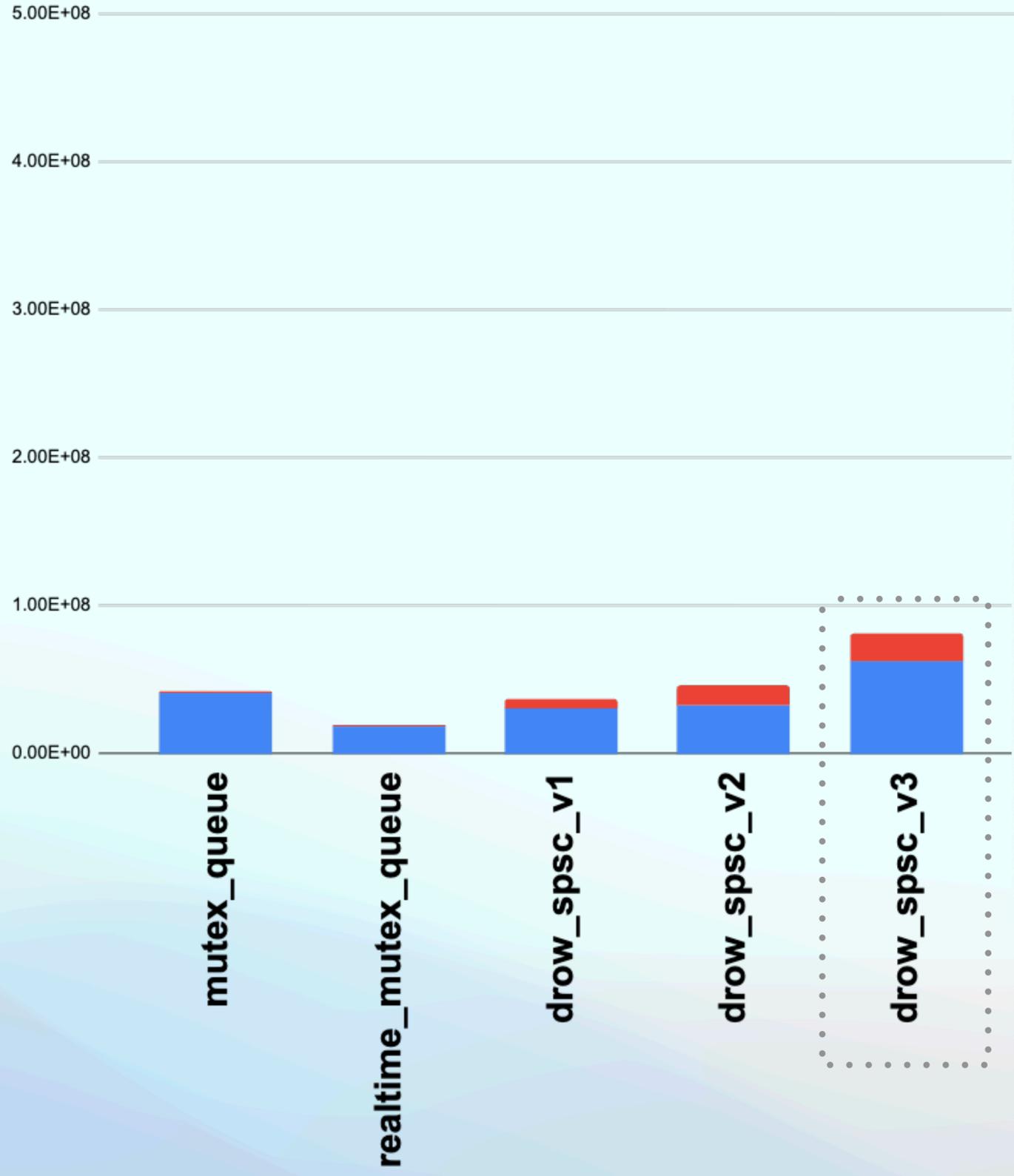
```

bool try_pop (T& v)
{
    size_t current_head = head.load(std::memory_order_relaxed);

    if (current_head == tail.load(std::memory_order_acquire))
        return false;

    v = data[current_head];
    head.store (next_index(current_head), std::memory_order_release);
    return true;
}

```



```

template<typename T>
class drow_queue_v4
{
public:
    drow_queue_v4 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {}

    bool try_push (const T& v)
    {
        size_t current_tail = tail.load (std::memory_order_relaxed);
        size_t current_head = head.load (std::memory_order_acquire);

        size_t size = current_tail - current_head;

        if (size >= (capacity - 1)) // full
            return false;

        size_t index = current_tail & (capacity - 1);
        data[index] = v;
        tail.store (current_tail + 1, std::memory_order_release);

        return true;
    }

    bool try_pop (T& v)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t current_tail = tail.load (std::memory_order_acquire);

        if (current_head == current_tail) // empty
            return false;

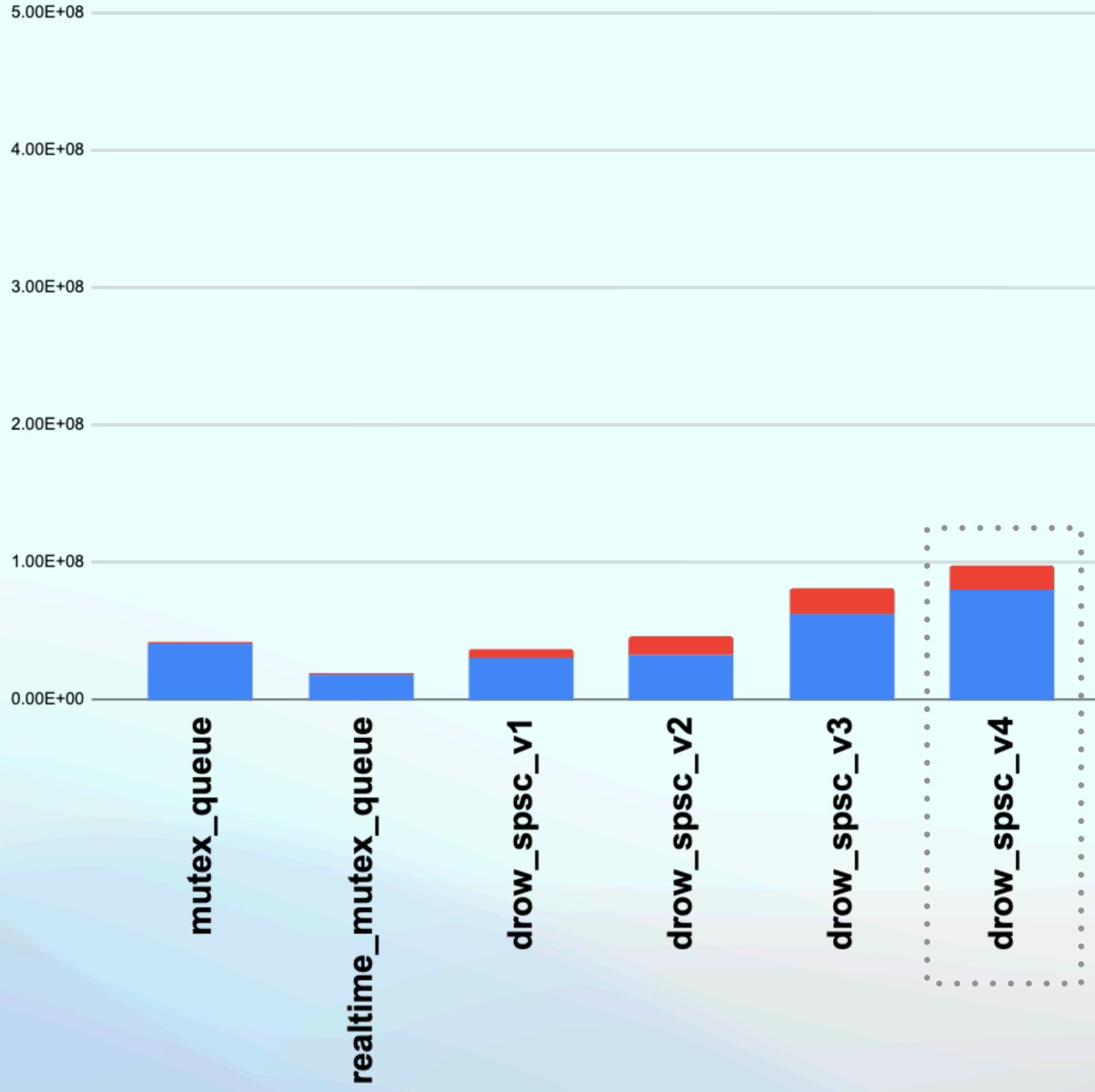
        size_t index = current_head & (capacity - 1);
        v = data[index];
        head.store (current_head + 1, std::memory_order_release);

        return true;
    }

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    std::atomic<size_t> head { 0 }, tail { 0 };
};

```

- Power of two & bit mask
- Monotonic head/tail counter
- Memory ordering



# Optimisations

## False Sharing

- When one or more variables share the same cache line
- The CPU will effectively lock access to the memory until it has been used by a single core
- Head/tail counters

```

template<typename T>
class drow_queue_v5
{
public:
    drow_queue_v5 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {}

    bool try_push (const T& v)
    {
        size_t current_tail = tail.load (std::memory_order_relaxed);
        size_t current_head = head.load (std::memory_order_acquire);

        size_t size = current_tail - current_head;

        if (size >= (capacity - 1)) // full
            return false;

        size_t index = current_tail & (capacity - 1);
        data[index] = v;
        tail.store (current_tail + 1, std::memory_order_release);

        return true;
    }

    bool try_pop (T& v)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t current_tail = tail.load (std::memory_order_acquire);

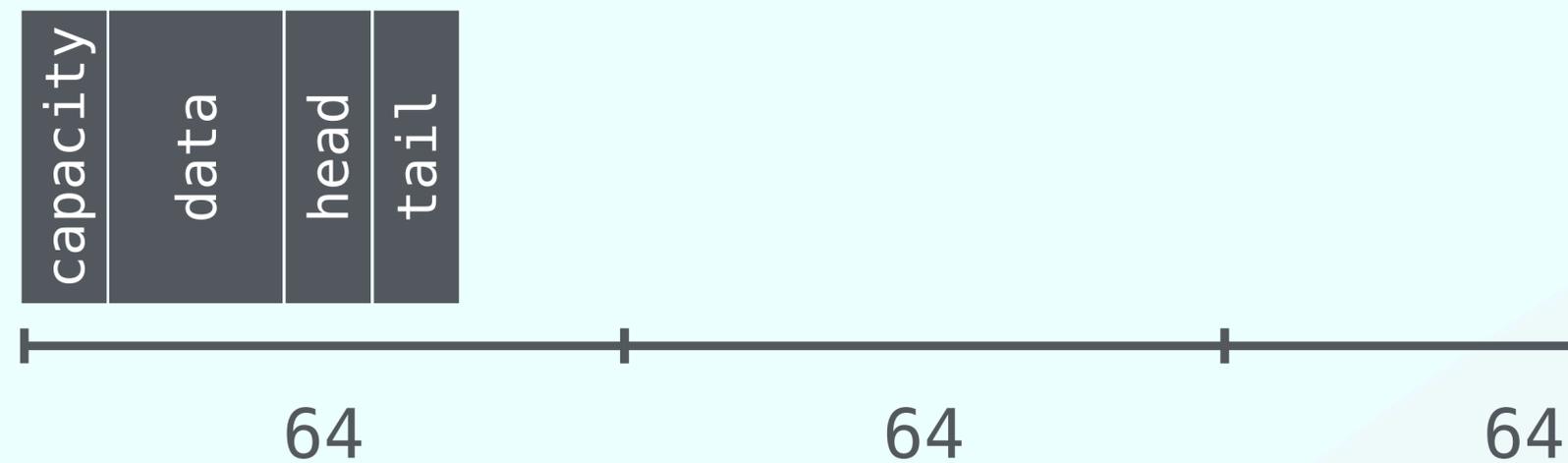
        if (current_head == current_tail) // empty
            return false;

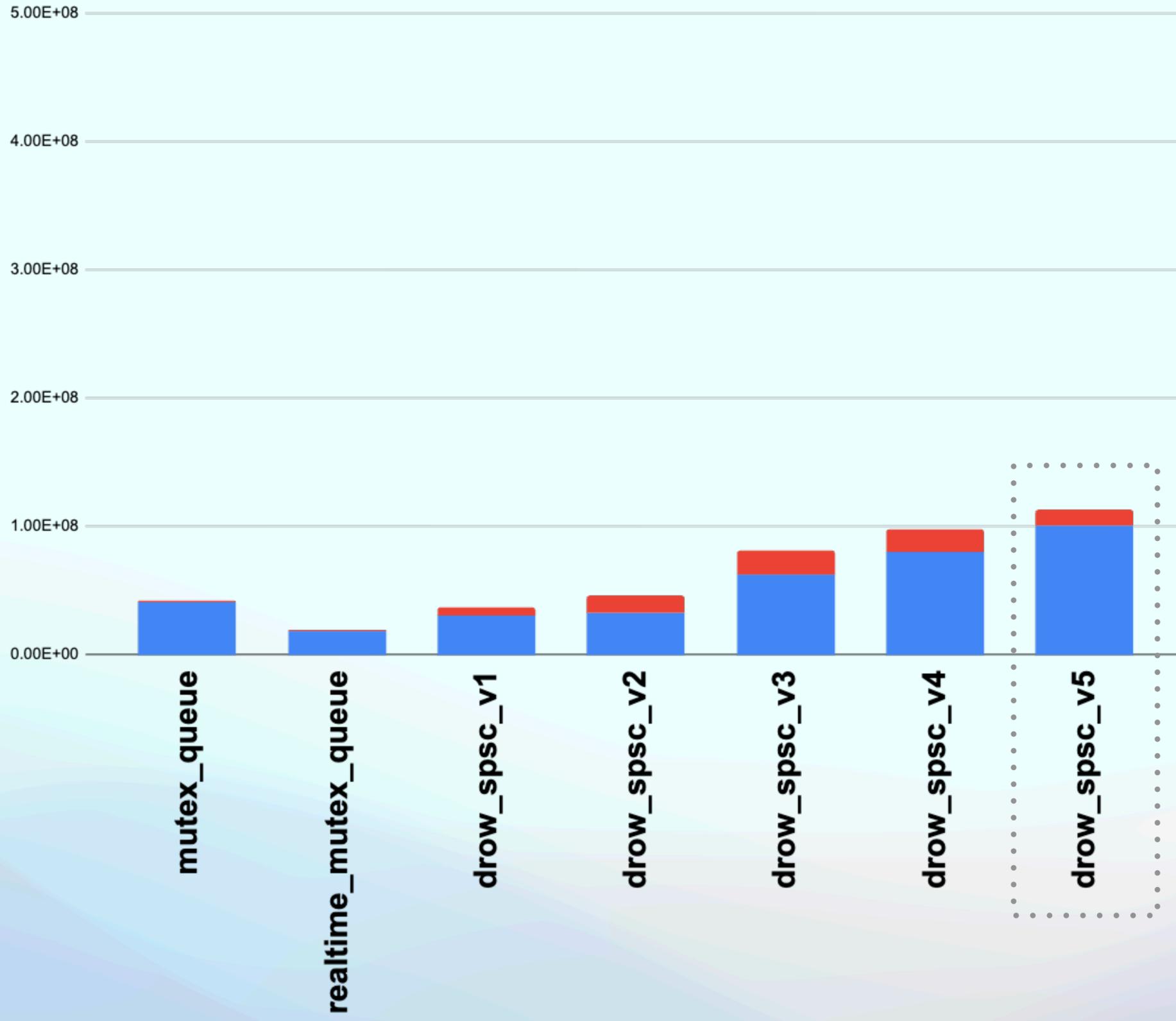
        size_t index = current_head & (capacity - 1);
        v = data[index];
        head.store (current_head + 1, std::memory_order_release);

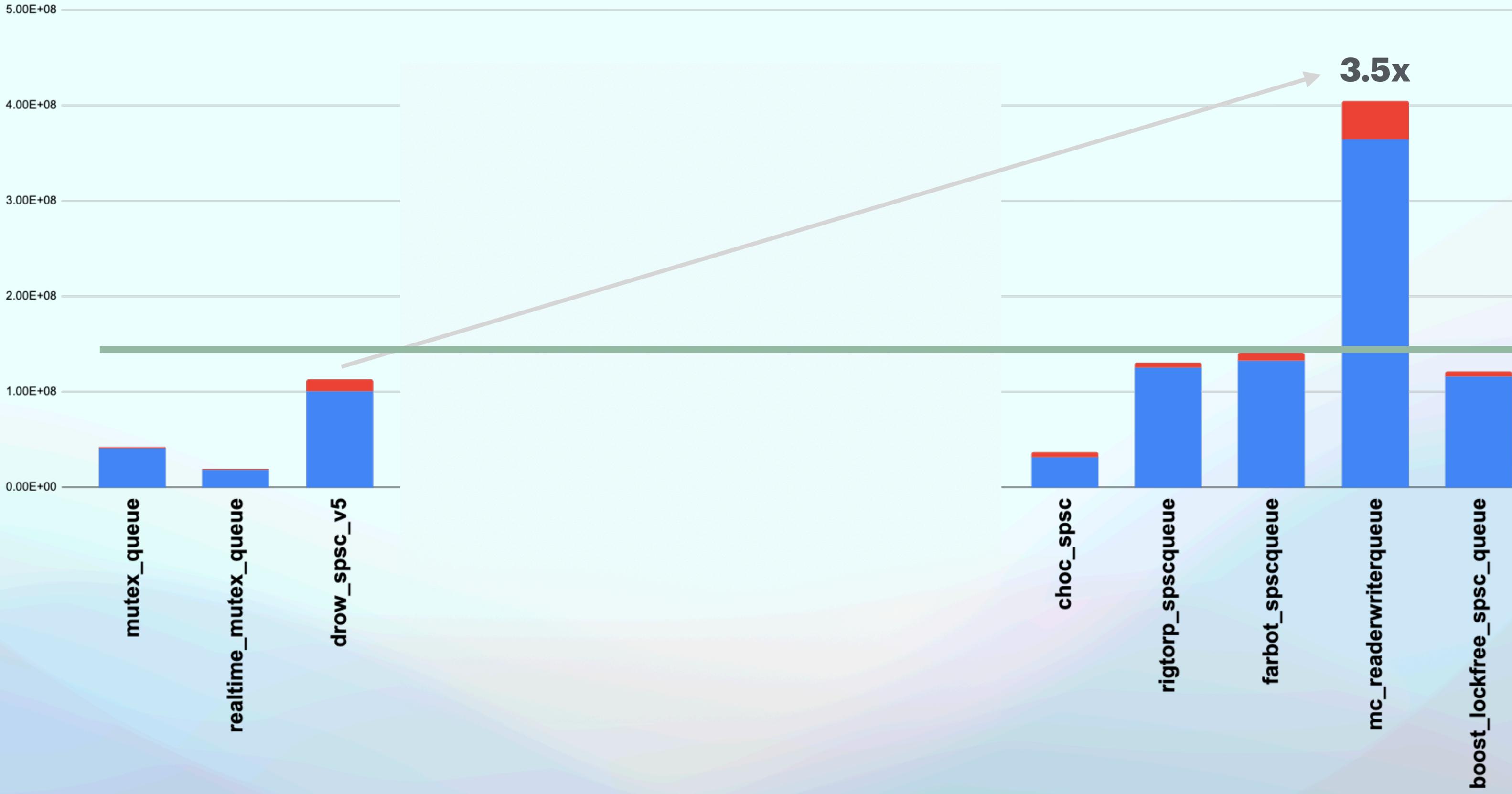
        return true;
    }

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };

```







```

template<typename T>
class drow_queue_v6
{
public:
    drow_queue_v6 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {}

```

```

bool try_push (const T& v)
{
    size_t current_tail = tail.load (std::memory_order_relaxed);

    size_t size = current_tail - cached_head;

    if (size >= (capacity - 1)) // full
    {
        cached_head = head.load (std::memory_order_acquire);
        size = current_tail - cached_head;

        if (size >= (capacity - 1))
            return false;
    }

    size_t index = current_tail & (capacity - 1);
    data[index] = v;
    tail.store (current_tail + 1, std::memory_order_release);

    return true;
}

```

```

bool try_pop (T& v)
{
    size_t current_head = head.load (std::memory_order_relaxed);

    if (current_head == cached_tail) // empty
    {
        cached_tail = tail.load (std::memory_order_acquire);

        if (current_head == cached_tail) // empty
            return false;
    }

    size_t index = current_head & (capacity - 1);
    v = data[index];
    head.store (current_head + 1, std::memory_order_release);

    return true;
}

```

5.00E+08  
4.00E+08  
3.00E+08  
2.00E+08  
1.00E+08  
0.00E+00

**mutex\_queue**

**realtime\_mutex\_queue**

**drow\_spsc\_v5**

**drow\_spsc\_v6**



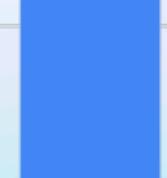
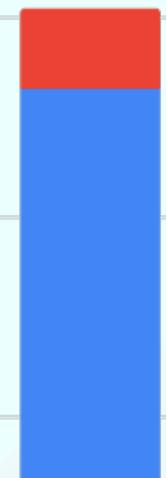
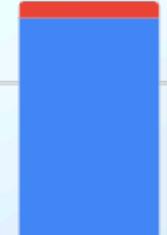
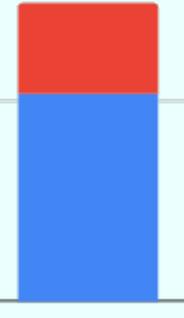
**choc\_spsc**

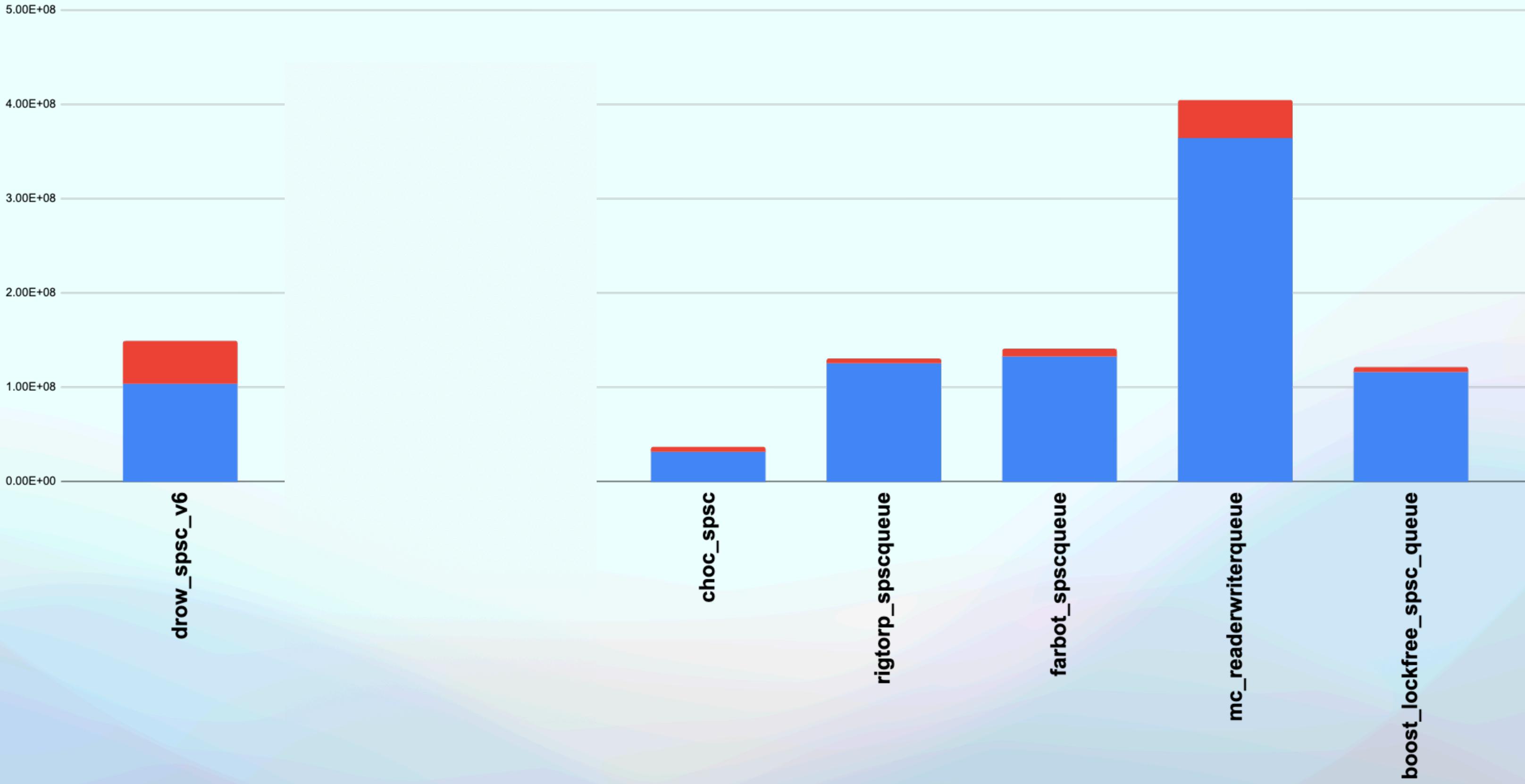
**rigtorp\_spscqueue**

**farbot\_spscqueue**

**mc\_readerwriterqueue**

**boost\_lockfree\_spsc\_queue**





```

template<typename T>
class drow_queue_v8
{
public:
    drow_queue_v8 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {}

    bool try_push (const T& v)
    {
        size_t current_tail = tail.load (std::memory_order_relaxed);
        size_t size = current_tail - cached_head;

        if (size >= (capacity_minus_one)) // full
        {
            cached_head = head.load (std::memory_order_relaxed);

            size = current_tail - cached_head;

            if (size >= (capacity_minus_one))
                return false;
        }

        size_t index = current_tail & (capacity_minus_one);
        data[index] = v;

        tail.store (current_tail + 1, std::memory_order_relaxed);

        return true;
    }

private:
    size_t capacity = 0;
    size_t capacity_minus_one = capacity - 1;
    alignas (hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas (hardware_destructive_interference_size) size_t cached_tail { 0 };
    alignas (hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    alignas (hardware_destructive_interference_size) size_t cached_head { 0 };
    std::vector<T> data { std::vector<T> (capacity) };
};

```

```

bool try_pop (T& v)
{
    size_t current_head = head.load (std::memory_order_relaxed);

    if (current_head == cached_tail) // empty
    {
        cached_tail = tail.load (std::memory_order_relaxed);

        if (current_head == cached_tail) // empty
            return false;
    }

    size_t index = current_head & (capacity_minus_one);
    v = data[index];

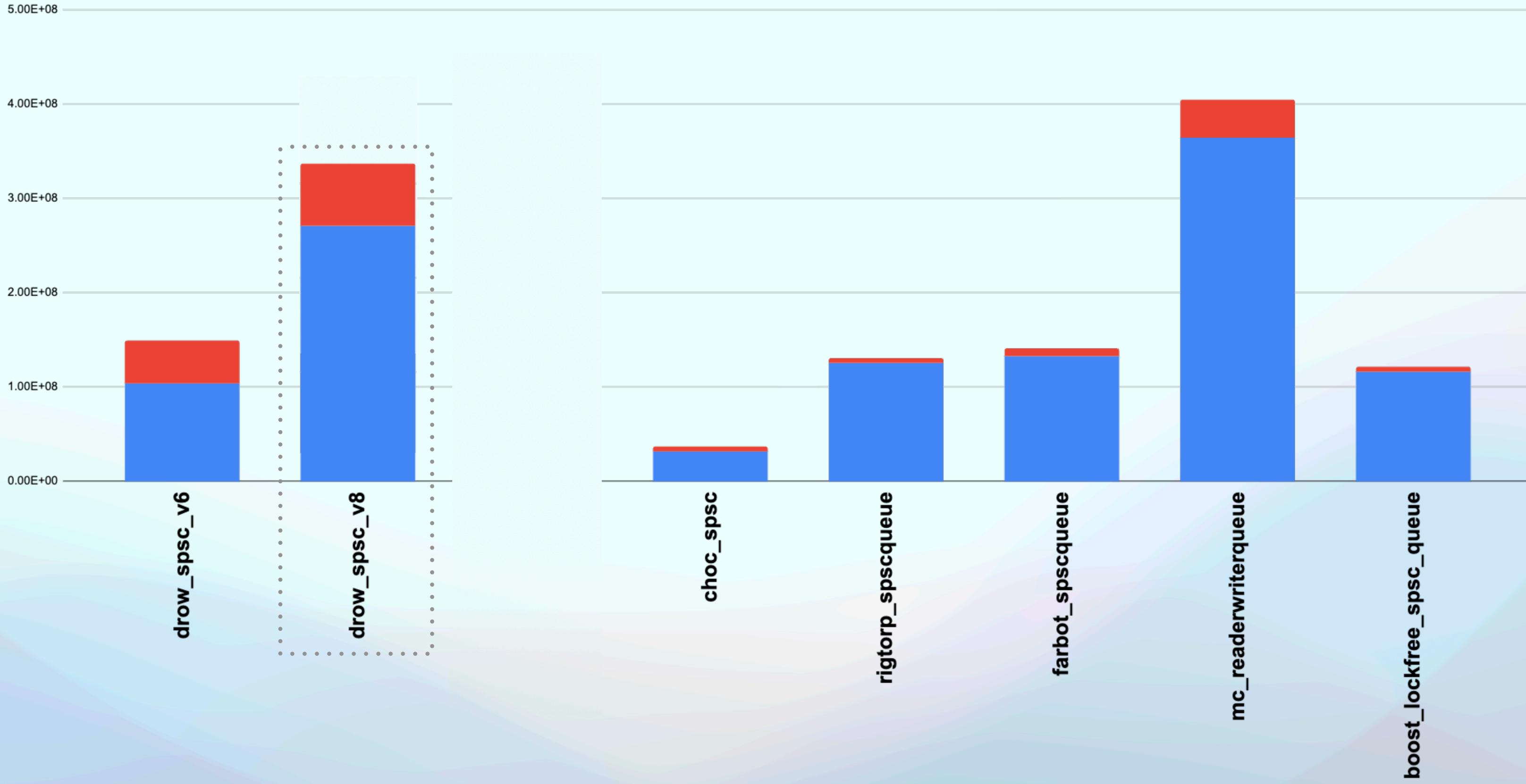
    head.store (current_head + 1, std::memory_order_relaxed);

    return true;
}

```

### 3 Changes:

1. `head/tail` relaxed load/stores  
Accessed concurrently, need to be atomic
2. Fences in `try_push` ensure store to `data` isn't reordered before the the store to `tail` or before the load from `head`
3. Fences in `try_pop` ensure that the read from `data` isn't reordered before the load from `tail` or after the store to `head`



A Note on AI...

does this code have any soundness issues?

### **Summary**

*The code is sound (it will work correctly), but the acquire fence in the `else` branch is redundant and can be removed. The synchronization is already established by the acquire fence that was executed when `cached_tail` was originally loaded.*

but if the queue is not empty according to the `cached_tail`, then won't any previous writes to the data be stale? The fence in the `else` branch was to catch that scenario

<snip>

4. When you now read `data[current_head]` where `current_head < 10`, you're reading a slot that was written even earlier than `data[9]`, so it's definitely visible.

### **Wait, is that actually true?**

Actually, you've identified a real concern! Let me reconsider...

<snip>

Actually, I think you're right! Without the fence in the `else` branch, you might read stale data if:

You cached `tail = 100`

Producer wrote items 100-200

You try to pop item 50 (from earlier)

But wait... you can't pop item 50 because `head` would be at least 100...

Hmm, let me think about this more carefully...

### **Actually, Here's Why It IS Sound**

<snip>

The acquire fence when loading `tail` establishes happens-before with all prior pushes, not just the most recent one. So you're guaranteed to see data writes for all indices less than `cached_tail`.

**Your code is sound.** The `else` fence is indeed redundant because the synchronization from the `cached tail` load covers all the data you might read while `current_head < cached_tail`.

# Opus 4.6 Extended Thinking:

## Scrutinized synchronization patterns

The fence pattern is sound for SPSC. The relaxed store preceded by a release fence on the writer side, paired with a relaxed load followed by an acquire fence on the reader side, satisfies the synchronization requirements of the standard (§29.8 / §33.5.11 in C++20). The fast-path elision of the fence is safe because the cached value was established under a prior synchronization, and the data for those slots was already made visible.



## Scrutinized synchronization patterns and identified off-by-one capacity bug

Let me analyze this SPSC (Single Producer Single Consumer) queue for soundness issues.

Several issues:

1. **Member initialization order bug:** `capacity_minus_one` is initialized as `capacity - 1`, but `capacity` is set in the constructor body... wait, no. `capacity` is initialized in the member initializer list. Let me re-read.

Actually, `capacity` has a default member initializer of `0`, but the constructor initializer list sets it to `std::bit_ceil(capacity_)`. Members are initialized in declaration order. So `capacity` gets initialized first to `std::bit_ceil(capacity_)`, then `capacity_minus_one` gets initialized to `capacity - 1`. That should be fine actually since they're declared in that order.

The vector is also initialized after `capacity` in declaration order, so it correctly uses the computed capacity value.

2. The capacity check uses `capacity_minus_one` but should it be `capacity`? The check is `size >= capacity_minus_one`. For a queue with capacity N (power of 2), you typically can store N-1 elements to distinguish full from empty. But with separate head/tail counters that wrap around, this implementation doesn't need to waste a slot—the full condition should actually be `size >= capacity`, not `size >= capacity_minus_one`. This looks like a bug.

Actually, let me think through this more carefully. If capacity is 8, then `capacity_minus_one` is 7 and the mask is also 7. With up to 7 elements, head and tail indices differ by at most 7, and with a mask of 7, all indices map to distinct slots. If we allowed 8 elements instead, head could equal tail modulo capacity, but since we're using unwrapped counters, `tail - head == 8` is distinguishable from `tail - head == 0`. So allowing full capacity would actually be sound with this design—using `capacity_minus_one` just unnecessarily wastes one slot.

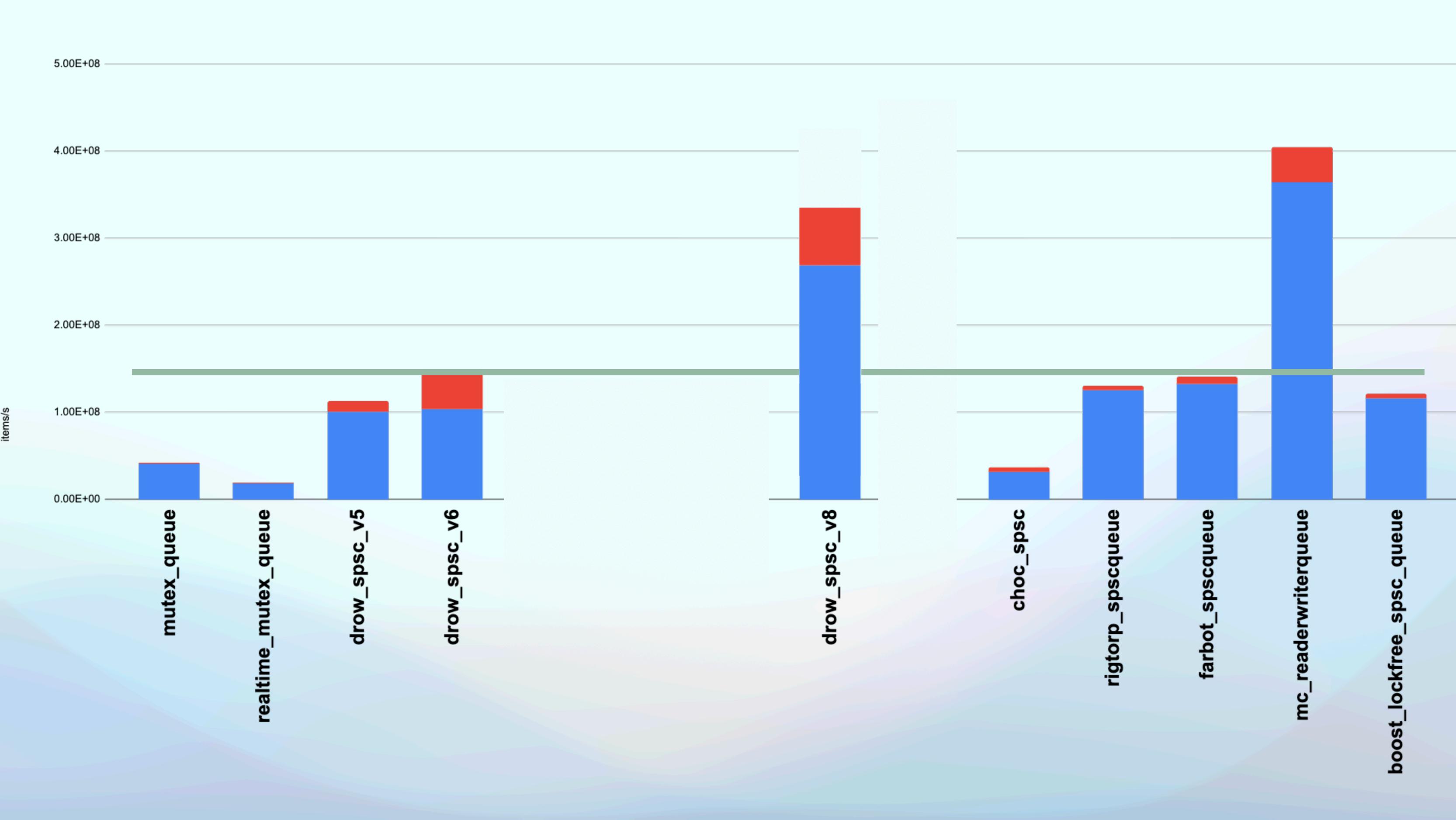
Wait, let me reconsider...

Now I'm examining the fence ordering in `try_pop`. The release fence before storing `head` seems problematic. The consumer needs an acquire fence to see data written by the producer, which it gets from the acquire fence after loading `tail`. But before advancing `head`, it needs a release fence to ensure the data read happens before the head is advanced, preventing the producer from overwriting that slot. The consumer reads the data first, then advances head, and the producer checks head to determine if there's space...

Actually, thinking through the C++ memory model more carefully: a release fence prevents preceding reads and writes from being reordered past subsequent stores, so the load of `data[index]` can't be reordered past the `head.store()`. That checks out. But on the producer side, when it loads `head` with relaxed ordering and then executes an acquire fence, it needs to ensure it sees the consumer's updated head value before proceeding.

Let me map out both paths more carefully. The producer loads tail relaxed, checks against its cached head, and if the queue is full it reloads head with relaxed ordering followed by an acquire fence, then writes to data, executes a release fence, and stores tail relaxed. The consumer mirrors this: it loads head relaxed, checks against cached tail, reloads tail with relaxed ordering plus acquire fence if empty, reads from data, executes a release fence, and stores head relaxed.

The synchronization point is when the producer writes data and issues a release fence before storing tail — this synchronizes-with the consumer's acquire fence after loading tail, ensuring the consumer sees the written data before reading it.



```

if constexpr (queue_end == queue_end::producer)
{
    for (size_t i = 0; i < iters; ++i)
    {
        stopwatch iter_sw;

        if (queue.try_push(i))
        {
            sum += i;
            stats.add_result (iter_sw.get());
            ++res.num_valid;
        }
        else
        {
            ++res.num_failed;
        }
    }
}

```

```

else
{
    for (size_t i = 0; i < iters; ++i)
    {
        stopwatch iter_sw;

        int v;
        if (queue.try_pop (v))
        {
            sum += v;
            ++res.num_valid;
            stats.add_result (iter_sw.get());
        }
        else
        {
            ++res.num_failed;
        }
    }
}

```

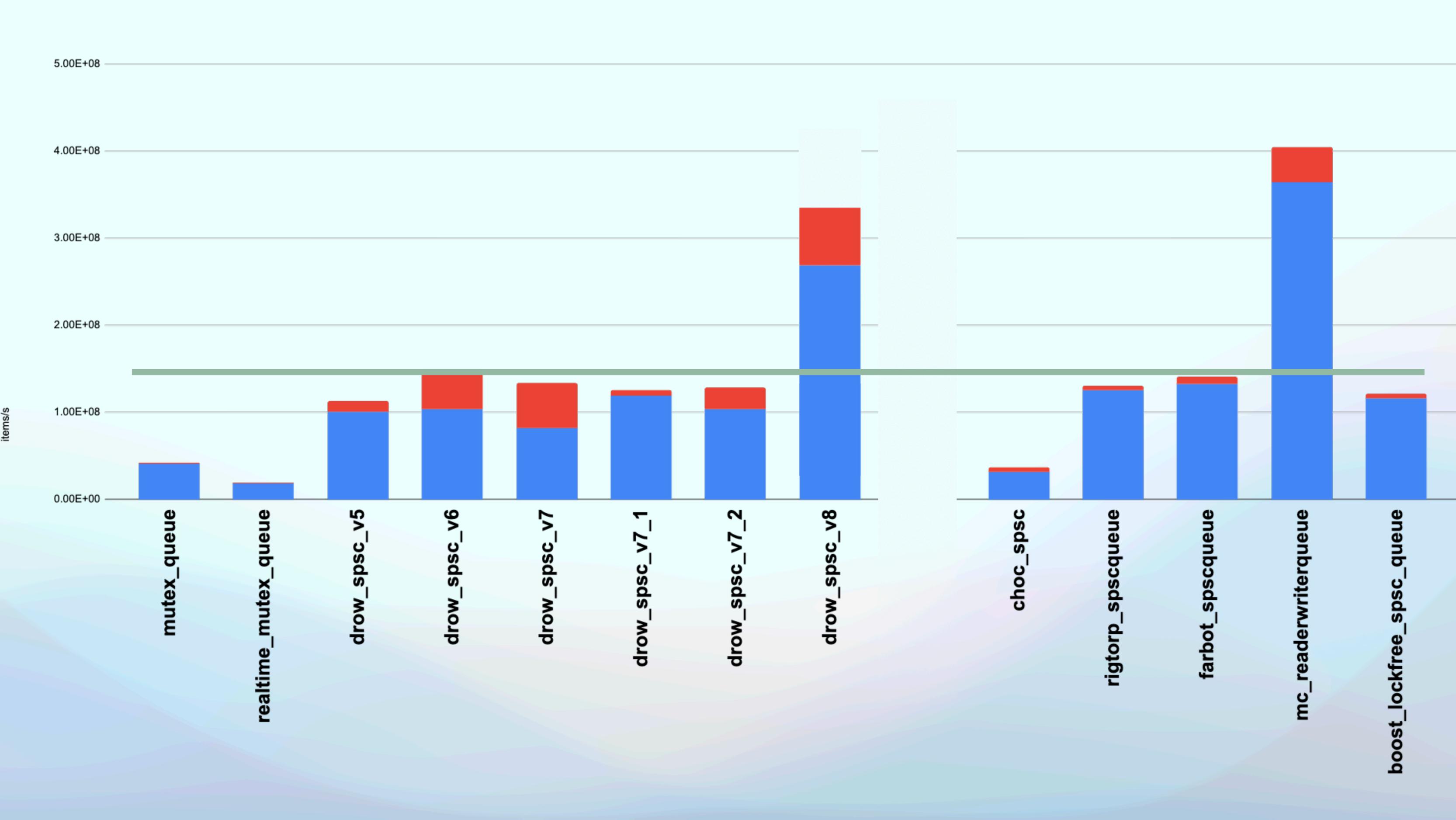
```

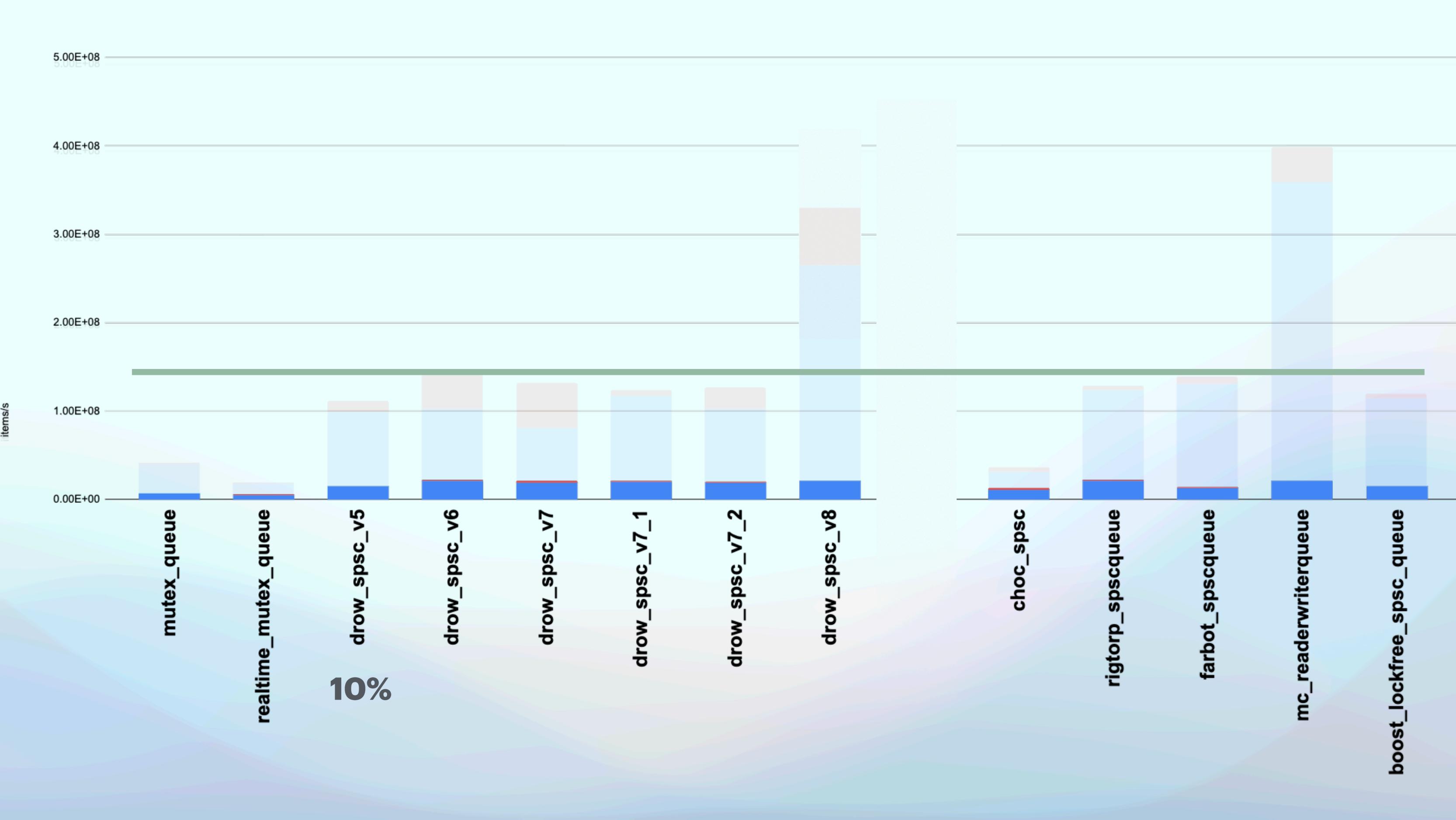
inline void time_statistics::add_result
(double secondsElapsed)
{
    if (numRuns == 0)
    {
        max = secondsElapsed;
        min = secondsElapsed;
    }
    else
    {
        max = std::max (max, secondsElapsed);
        min = std::min (min, secondsElapsed);
    }

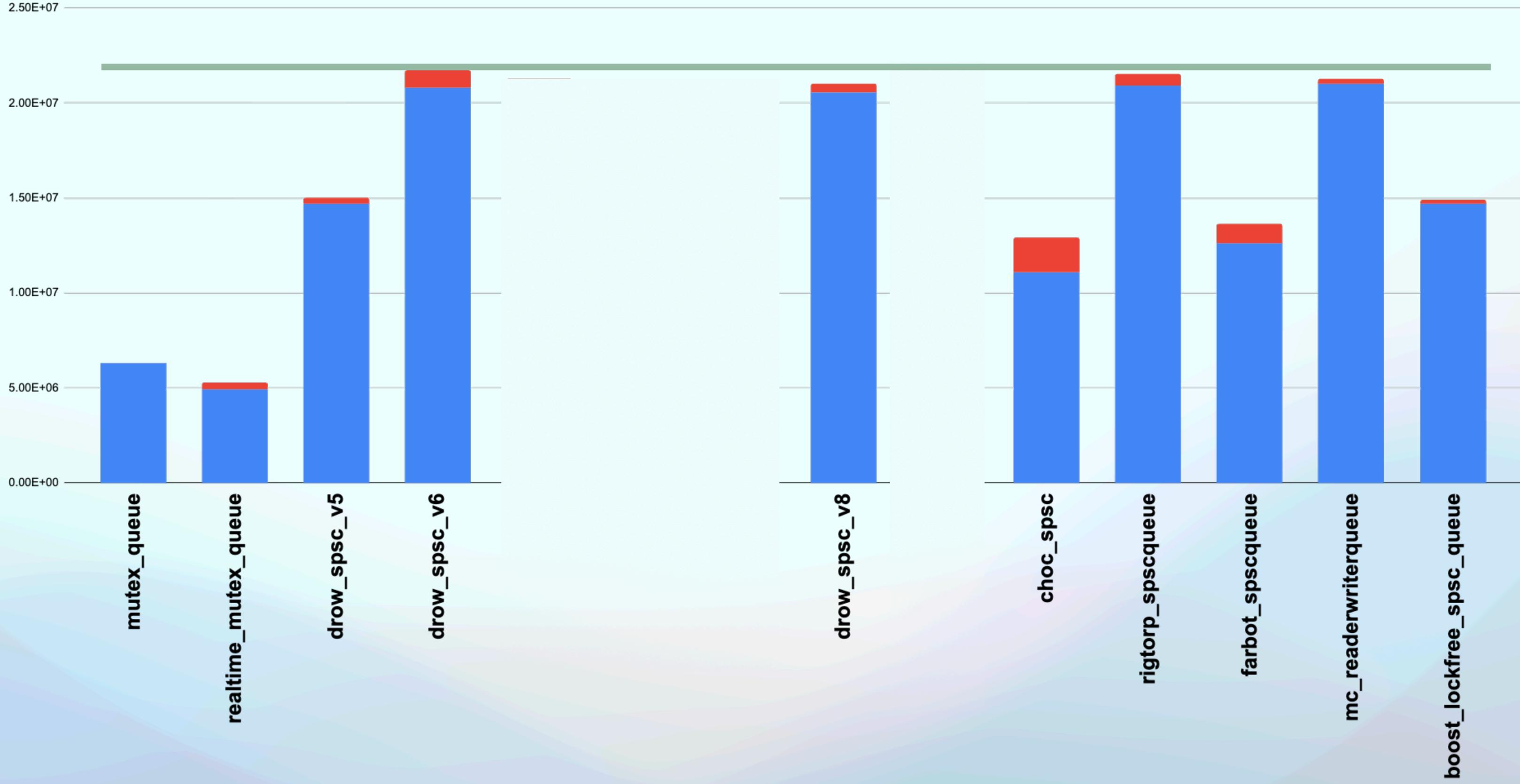
    ++numRuns;
    total += secondsElapsed;

    // Seconds
    {
        const double delta = secondsElapsed - mean;
        mean += delta / (double) numRuns;
        const double delta2 = secondsElapsed - mean;
        m2 += delta * delta2;
    }
}

```







LEVEL 1





# Single Producer Multi Consumer SPMC

- Command queue
- Worker threads
- Visualisers
- Networking
- MIDI

# Advanced Behaviour

## MPSC/MPMC

<b>push</b>	Block on full*	Overwrite when full	Fail/try_
<b>pop</b>	Block on empty*	Return default on empty	Fail/try_
<b>capacity</b>	Static (compile-time)	Fixed (runtime)	Dynamic (grows)
<b>Bulk push/pop</b>	No - single item	Yes - multiple items	
<b>Message size</b>	Fixed	Dynamic	
<b>Gurantees</b>	None - blocking	Lock-free	Wait-free
<b>Message size limit</b>	Limited (8 bytes?)	Unlited	
<b>Triviality</b>	Trival	Non-trivial	
<b>Num processes</b>	Single	Inter-process	
<b>Threads</b>	Multiple-producers	Multiple-consumers	<i>Max-num threads</i>
<b>Serialisation</b>	Strict global order	Relaxed	

# Advanced Behaviour

## MPSC/MPMC

<b>push</b>	Block on full*	Overwrite when full	Fail/try_
<b>pop</b>	Block on empty*	Return default on empty	Fail/try_
<b>capacity</b>	Static (compile-time)	Fixed (runtime)	Dynamic (grows)
<b>Bulk push/pop</b>	No - single item	Yes - multiple items	
<b>Message size</b>	Fixed	Dynamic	
<b>Gurantees</b>	None - blocking	Lock-free	Wait-free
<b>Message size limit</b>	Limited (8 bytes?)	Unlimgted	
<b>Triviality</b>	Trival	Non-trivial	
<b>Num processes</b>	Single	Inter-process	
<b>Threads</b>	Multiple-producers	Multiple-consumers	<i>Max-num threads</i>
<b>Serialisation</b>	Strict global order	Relaxed	

```

template<typename T>
class drow_spmc_v5
{
public:
    drow_spmc_v5 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {}

    bool try_push (const T& v)
    {
        size_t current_tail = tail.load (std::memory_order_relaxed);
        size_t current_head = head.load(std::memory_order_acquire);

        size_t size = current_tail - current_head;

        if (size >= (capacity - 1)) // full
            return false;

        size_t index = current_tail & (capacity - 1);
        data[index] = v;
        tail.store (current_tail + 1, std::memory_order_release);

        return true;
    }

private:
    size_t capacity = 0;
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<T> data { std::vector<T> (capacity) };
};

```

```

bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t current_tail = tail.load (std::memory_order_acquire);

        if (current_head == current_tail) // empty
            return false;

        size_t index = current_head & (capacity - 1);

        // Try to claim this slot atomically
        if (head.compare_exchange_weak(current_head, current_head + 1,
                                       std::memory_order_release,
                                       std::memory_order_relaxed))
        {
            // Successfully claimed the slot
            v = data[index];
            return true;
        }

        // CAS failed, another consumer claimed it. Retry.
    }
}

```

```
bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t current_tail = tail.load (std::memory_order_acquire);

        if (current_head == current_tail) // empty
            return false;

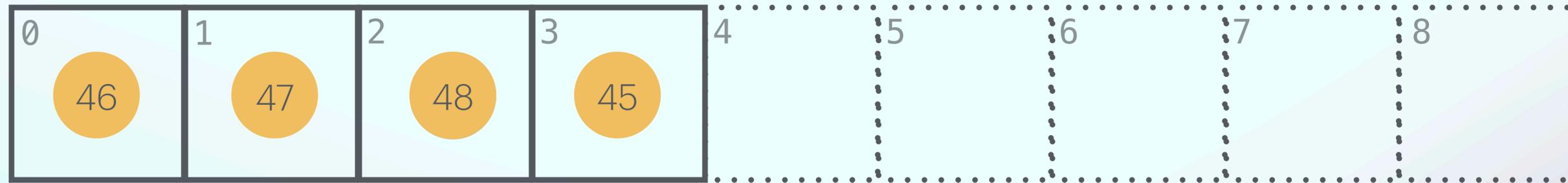
        size_t index = current_head & (capacity - 1);

        // Try to claim this slot atomically
        if (head.compare_exchange_weak(current_head, current_head + 1,
                                       std::memory_order_release,
                                       std::memory_order_relaxed))
        {
            // Successfully claimed the slot
            v = data[index];
            return true;
        }

        // CAS failed, another consumer claimed it. Retry.
    }
}
```

$$\begin{aligned} \text{free: } & (\text{cap} - 1) - (\text{tail} - \text{head}) = \\ & (4 - 1) - (6 - 4) = \\ & 3 - 2 = \\ & \mathbf{1} \end{aligned}$$

tail/  
write



head/  
read



```

bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_pending = pending_head.load (std::memory_order_acquire);
        size_t current_tail = tail.load (std::memory_order_acquire);

        if (current_pending == current_tail) // empty
            return false;

        size_t index = current_pending & (capacity - 1);

        // Try to claim this slot atomically
        if (pending_head.compare_exchange_weak (current_pending, current_pending + 1,
                                                std::memory_order_acquire,
                                                std::memory_order_relaxed))
        {
            // Successfully claimed the slot, now read the data
            v = data[index];

            // Now we need to commit this read
            // Wait until it's our turn to commit (all previous reads have committed)
            for (;;)
            {
                size_t expected_committed = current_pending;
                if (committed_head.compare_exchange_weak(expected_committed,
                                                         current_pending + 1,
                                                         std::memory_order_release,
                                                         std::memory_order_relaxed))
                {
                    // Successfully committed
                    return true;
                }

                // If CAS failed, it means we're not next in line yet
                // Keep spinning until committed_head catches up to our position
            }
        }

        // CAS failed, another consumer claimed it. Retry.
    }
}

```

```
private:
```

```
size_t capacity = 0;
```

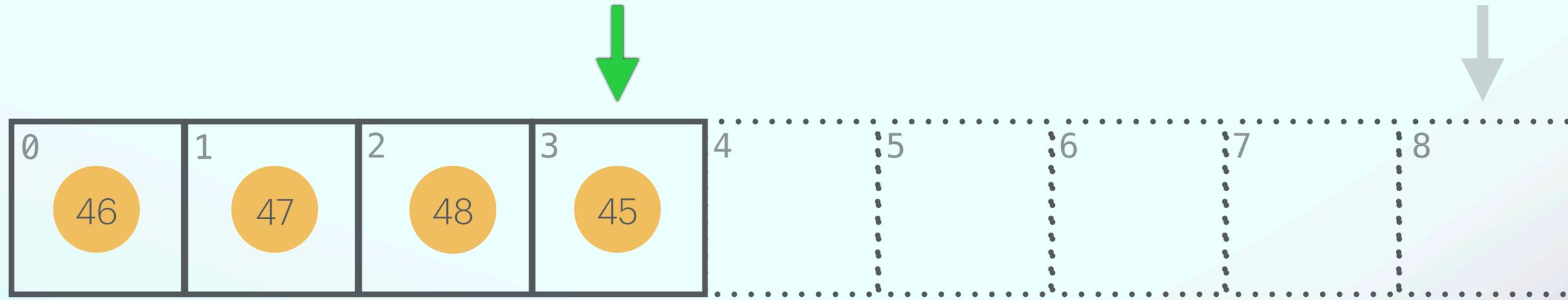
```
alignas(hardware_destructive_interference_size) std::atomic<size_t> pending_head { 0 };
```

```
alignas(hardware_destructive_interference_size) std::atomic<size_t> committed_head { 0 };
```

```
alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
```

```
std::vector<T> data { std::vector<T> (capacity) };
```

tail/  
write



head/  
read



1.00E+07

7.50E+06

5.00E+06

2.50E+06

0.00E+00

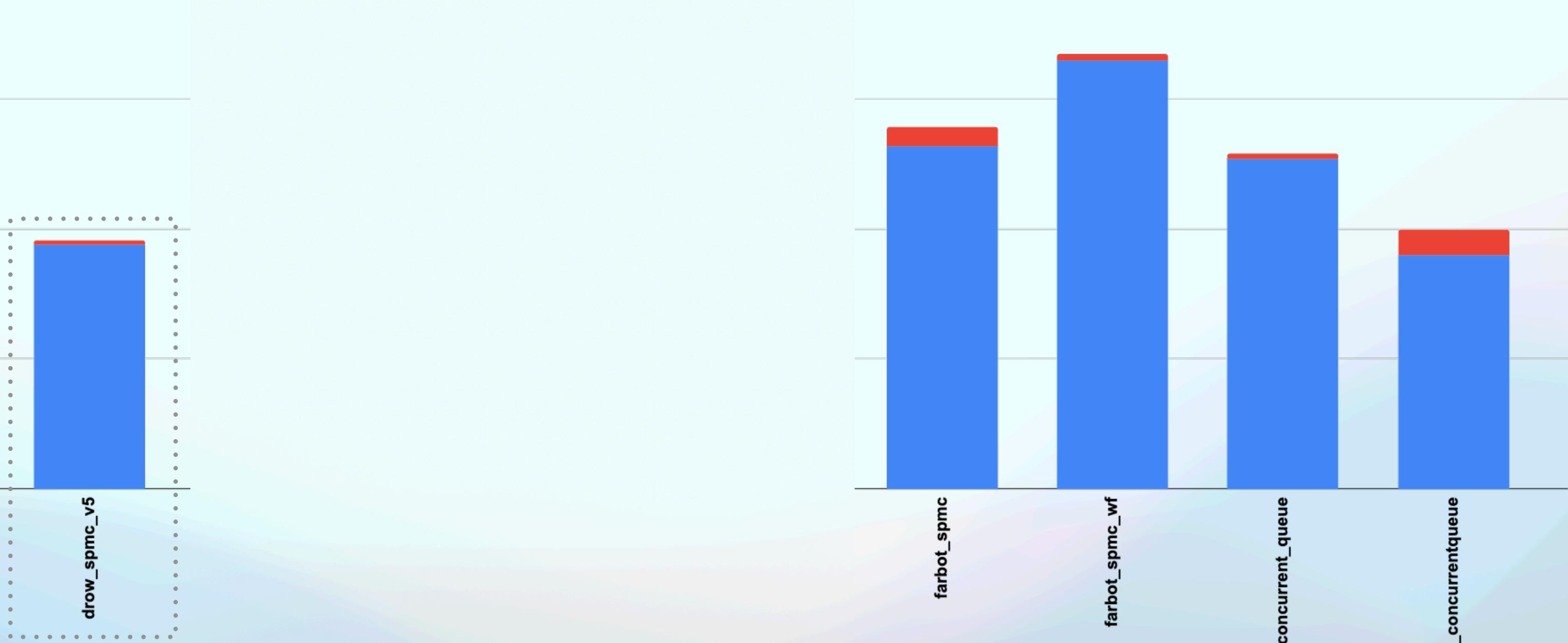
drow\_spmc\_v5

farbot\_spmc

farbot\_spmc\_wf

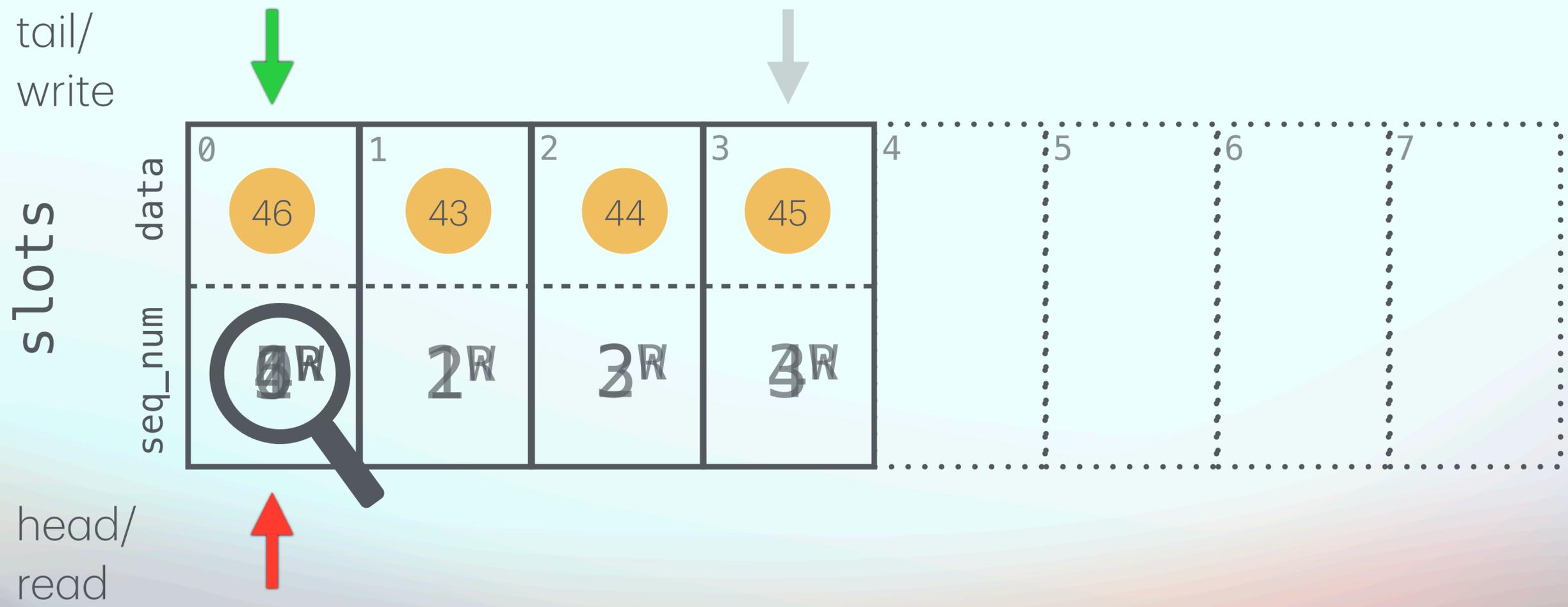
tbb\_concurrent\_queue

mc\_concurrentqueue



`seq_num == tail` → ready to write

`seq_num == head + 1` → ready to read



```

template<typename T>
class drow_spmc_v6
{
public:
    drow_spmc_v6 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        size_t current_tail = tail.load (std::memory_order_relaxed);
        size_t index = current_tail & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);

        if (size_t diff = seq - current_tail;
            diff == 0)
        {
            slot.data = v;

            // Mark slot as ready for reading
            slot.sequence.store (current_tail + 1, std::memory_order_release);
            tail.store (current_tail + 1, std::memory_order_relaxed);

            return true;
        }

        // Queue is full (slot hasn't been consumed yet)
        return false;
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas (hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas (hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots { std::vector<slot> (capacity) };
};

```

```

bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t index = current_head & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);
        ssize_t diff = seq - (current_head + 1);

        if (diff == 0) // Slot is ready for reading
        {
            // Try to claim this slot
            if (head.compare_exchange_weak (current_head, current_head + 1,
                                            std::memory_order_relaxed,
                                            std::memory_order_relaxed))
            {
                // Successfully claimed, read data
                v = slot.data;

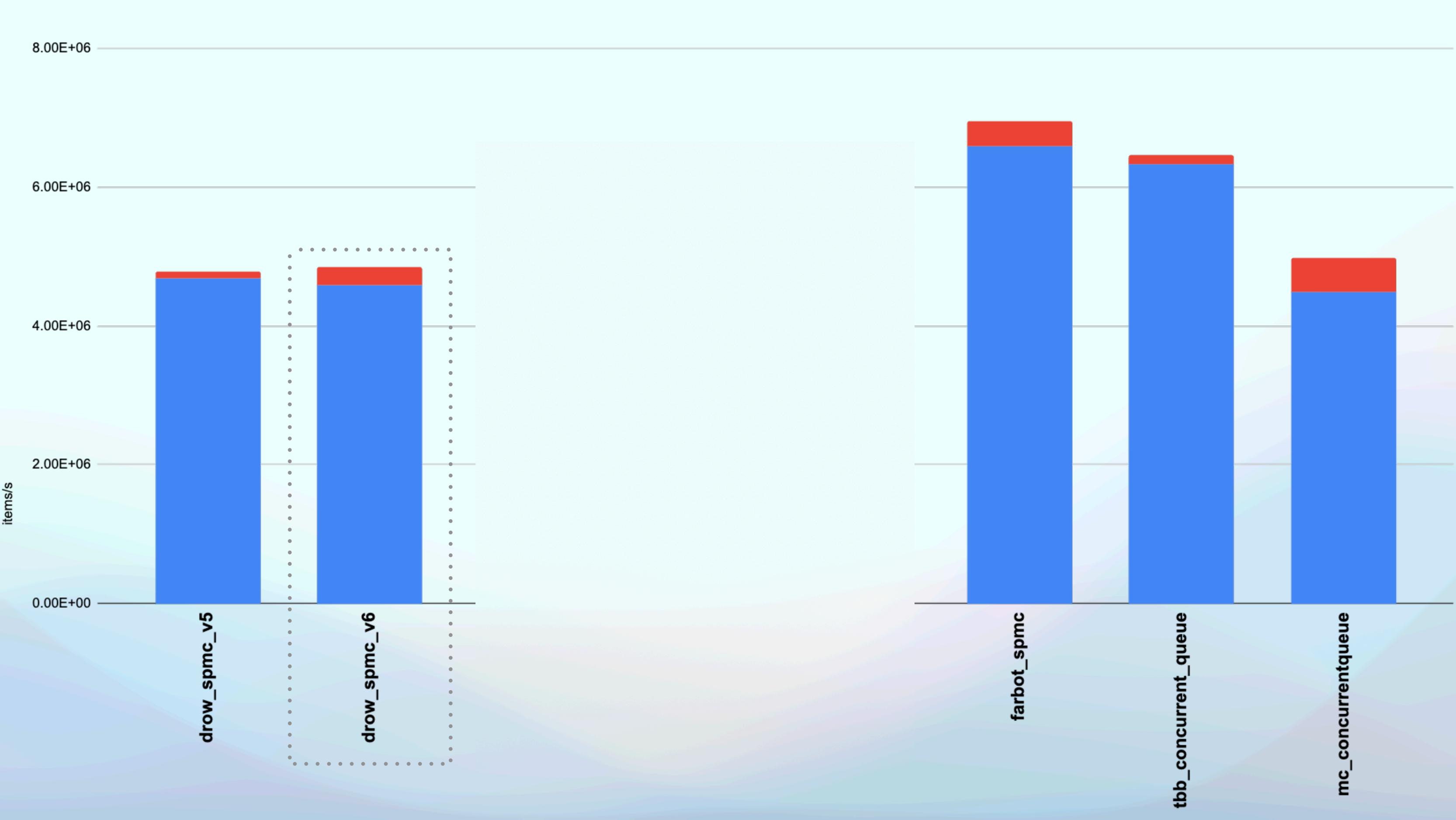
                // Mark slot as consumed and ready for writing
                slot.sequence.store (current_head + capacity, std::memory_order_release);

                return true;
            }

            // CAS failed, another consumer claimed it, retry
        }
        else if (diff < 0) // Queue is empty
        {
            return false;
        }

        // If diff > 0, slot isn't ready yet, retry
    }
}

```



```

template<typename T>
class drow_spmc_v7
{
public:
    drow_spmc_v7 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        size_t current_tail = cached_tail;
        size_t index = current_tail & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);

        if (size_t diff = seq - current_tail;
            diff == 0)
        {
            slot.data = v;

            // Mark slot as ready for reading
            cached_tail = current_tail + 1;
            slot.sequence.store (cached_tail, std::memory_order_release);
            tail.store (cached_tail, std::memory_order_relaxed);

            return true;
        }

        // Queue is full (slot hasn't been consumed yet)
        return false;
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas (hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas (hardware_destructive_interference_size) size_t cached_tail { 0 };
    alignas (hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots { std::vector<slot> (capacity) };
};

```

```

bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t index = current_head & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);
        ssize_t diff = seq - (current_head + 1);

        if (diff == 0) // Slot is ready for reading
        {
            // Try to claim this slot
            if (head.compare_exchange_weak (current_head, current_head + 1,
                                             std::memory_order_relaxed,
                                             std::memory_order_relaxed))
            {
                // Successfully claimed, read data
                v = slot.data;

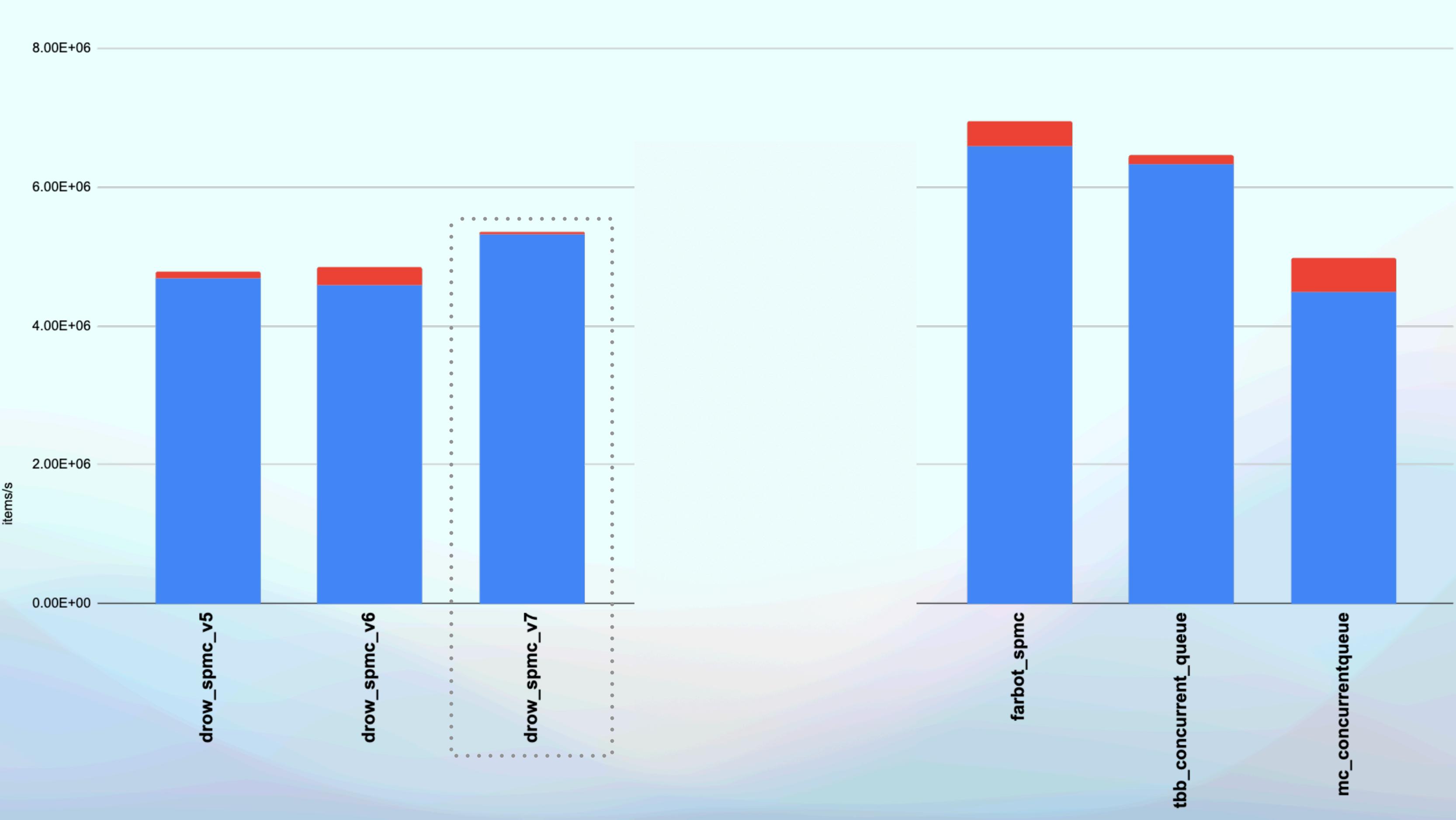
                // Mark slot as consumed and ready for writing
                slot.sequence.store (current_head + capacity, std::memory_order_release);

                return true;
            }

            // CAS failed, another consumer claimed it, retry
        }
        else if (diff < 0) // Queue is empty
        {
            return false;
        }

        // If diff > 0, slot isn't ready yet, retry
    }
}

```







```

template<typename T>
class drow_mpmc_v6
{
public:
    drow_mpmc_v6 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_)),
          slots (capacity)
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        for (;;)
        {
            size_t current_tail = tail.load (std::memory_order_relaxed);
            size_t index = current_tail & (capacity - 1);
            auto& slot = slots[index];

            size_t seq = slot.sequence.load (std::memory_order_acquire);

            if (ssize_t diff = seq - current_tail;
                diff == 0) // Slot is ready for writing
            {
                // Try to claim this slot
                if (tail.compare_exchange_weak (current_tail, current_tail + 1,
                                                std::memory_order_relaxed,
                                                std::memory_order_relaxed))
                {
                    // Successfully claimed, write data
                    slot.data = v;

                    // Mark slot as ready for reading
                    slot.sequence.store (current_tail + 1, std::memory_order_release);

                    return true;
                }

                // CAS failed, another producer claimed it, retry
            }
            else if (diff < 0) // Queue is full (slot hasn't been consumed yet)
            {
                return false;
            }

            // If diff > 0, slot isn't ready yet (another producer is writing), retry
        }
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots;
};

```

```

bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t index = current_head & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);

        if (ssize_t diff = seq - (current_head + 1);
            diff == 0) // Slot is ready for reading
        {
            // Try to claim this slot
            if (head.compare_exchange_weak (current_head, current_head + 1,
                                            std::memory_order_relaxed,
                                            std::memory_order_relaxed))
            {
                // Successfully claimed, read data
                v = slot.data;

                // Mark slot as consumed and ready for writing
                slot.sequence.store (current_head + capacity, std::memory_order_release);

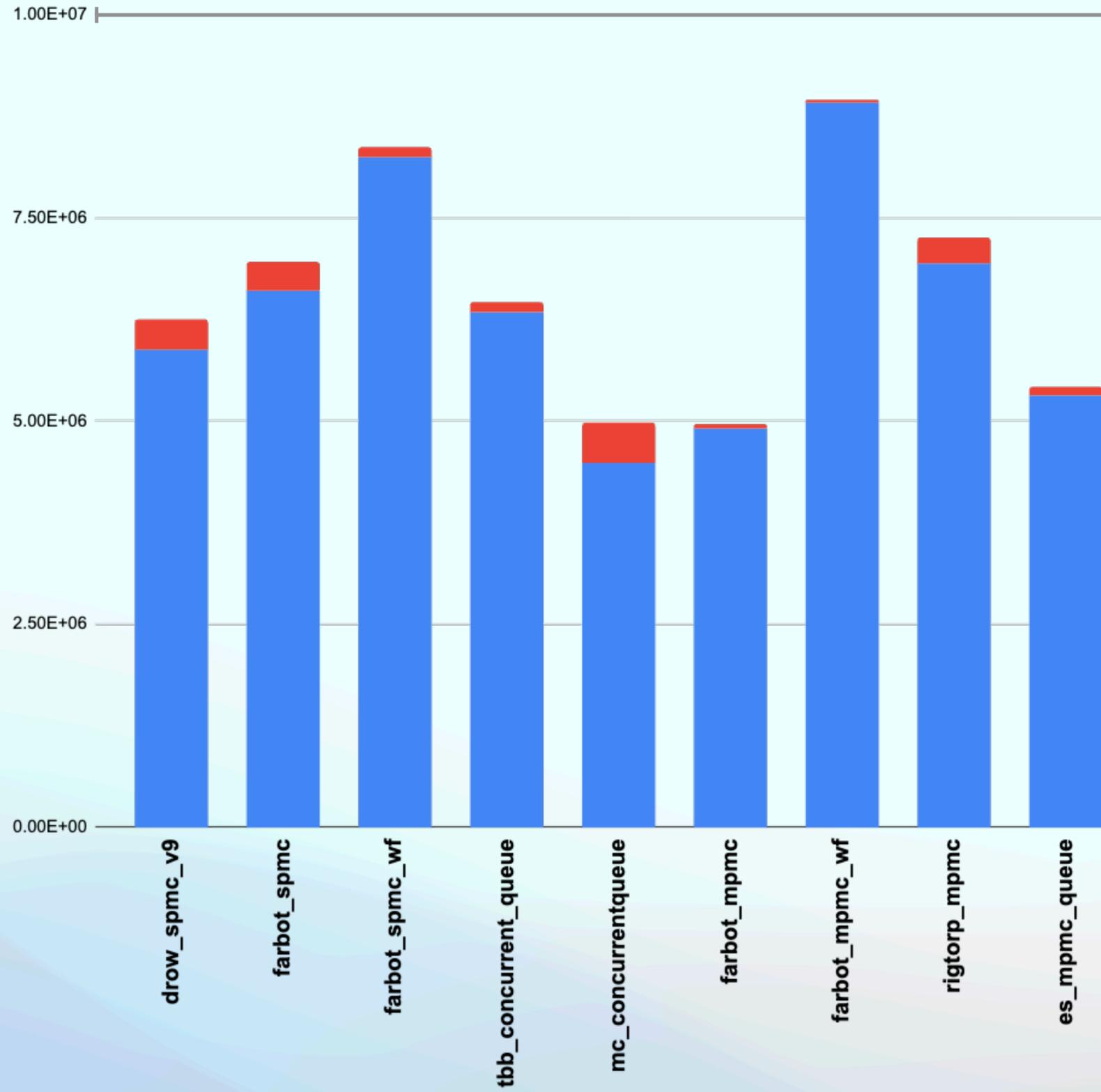
                return true;
            }

            // CAS failed, another consumer claimed it, retry
        }
        else if (diff < 0) // Queue is empty
        {
            return false;
        }

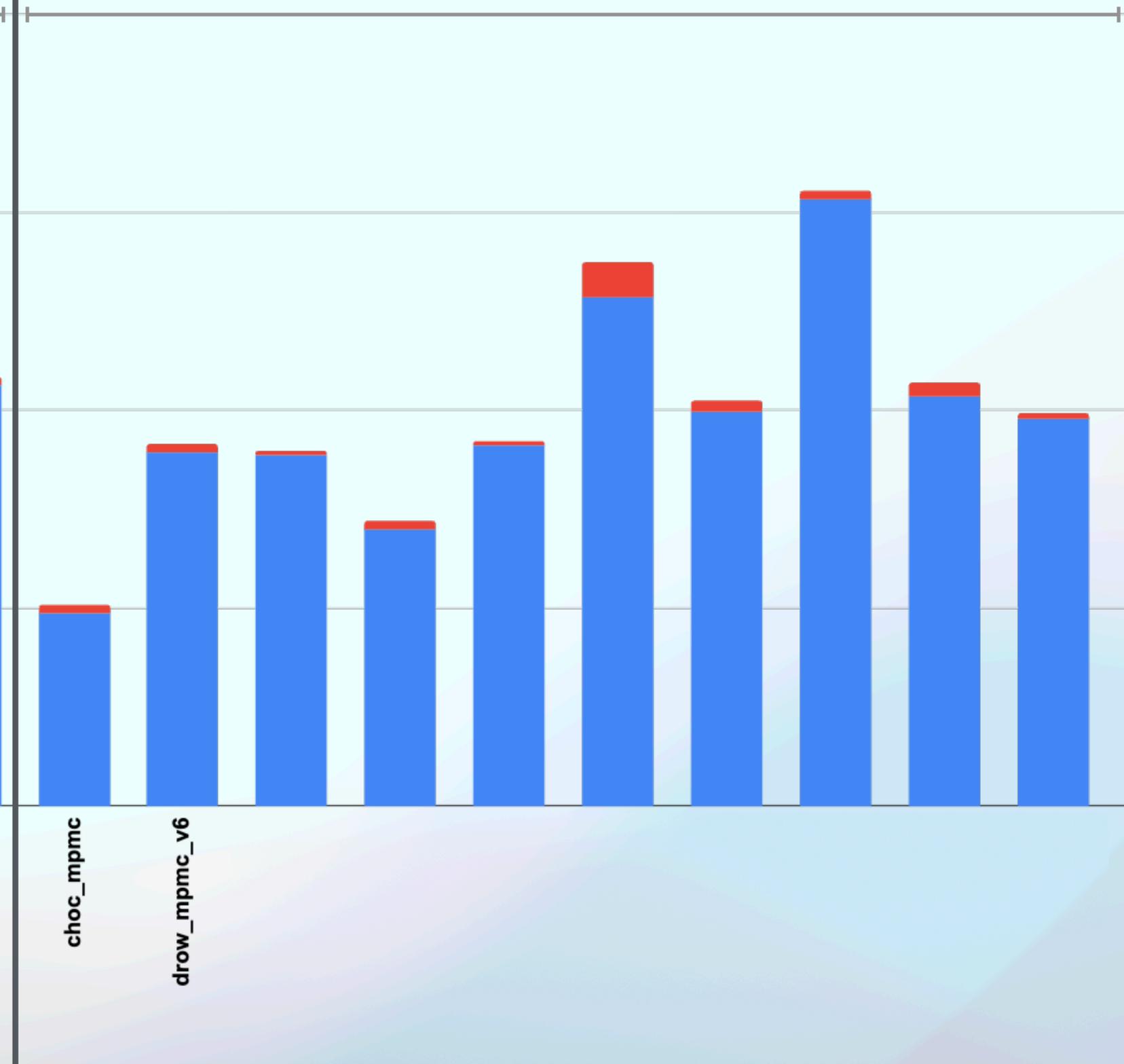
        // If diff > 0, slot isn't ready yet (another consumer is reading), retry
    }
}

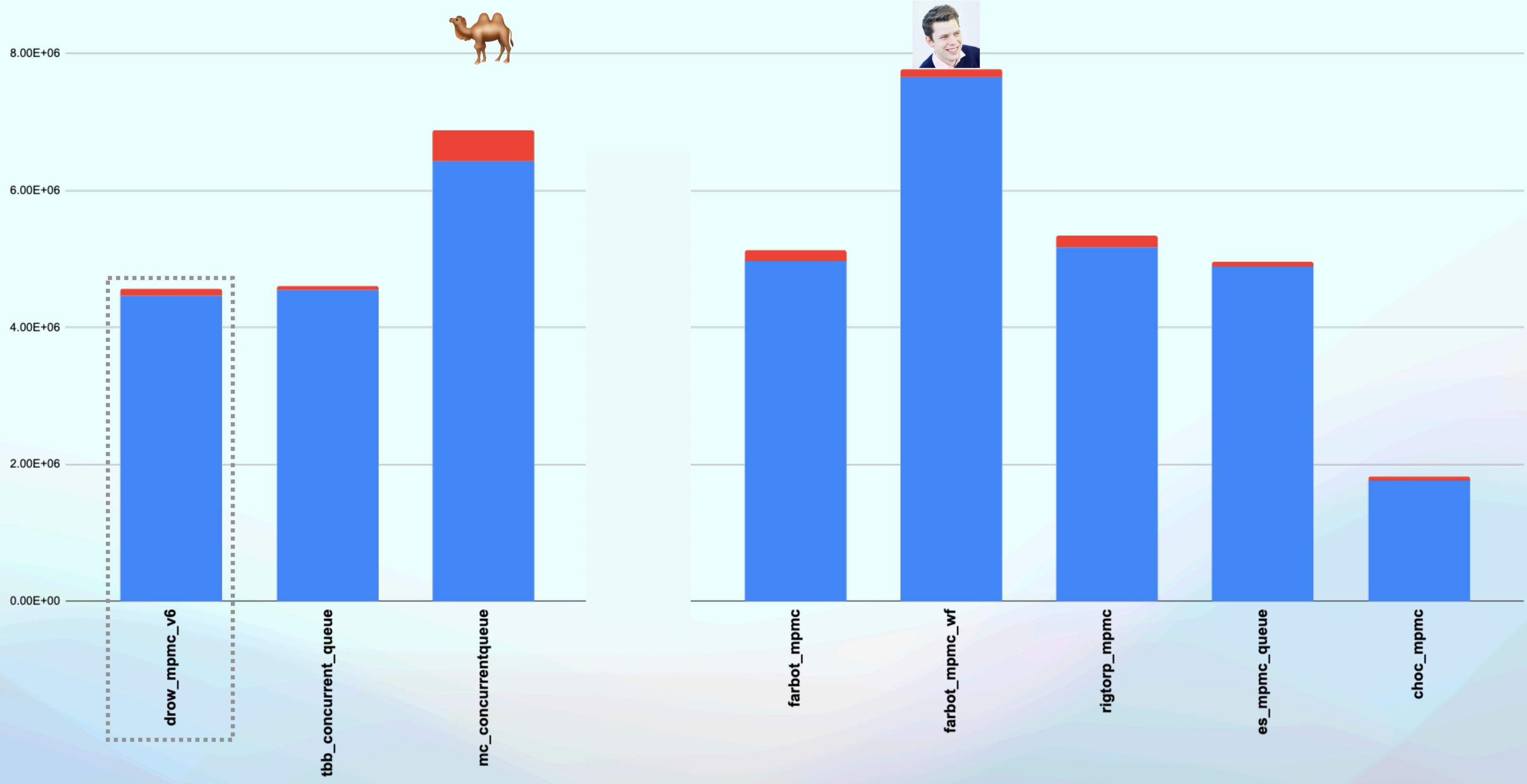
```

## Level 2: SPMC



## Level 3: MPMC





```

template<typename T>
class mc_concurrentqueue
{
public:
    mc_concurrentqueue (size_t capacity)
        : queue (capacity)
    {
    }

    bool try_push (const T& v)
    {
        return queue.try_enqueue (v);
    }

    bool try_pop (T& v)
    {
        return queue.try_dequeue (v);
    }

private:
    moodycamel::ConcurrentQueue<T> queue;
};

```

```

template<typename T>
class mc_concurrentqueue_sized
{
public:
    mc_concurrentqueue_sized (size_t capacity)
        : queue (capacity)
    {
    }

    bool try_push (const T& v)
    {
        if (! queue.try_enqueue (v))
            return false;

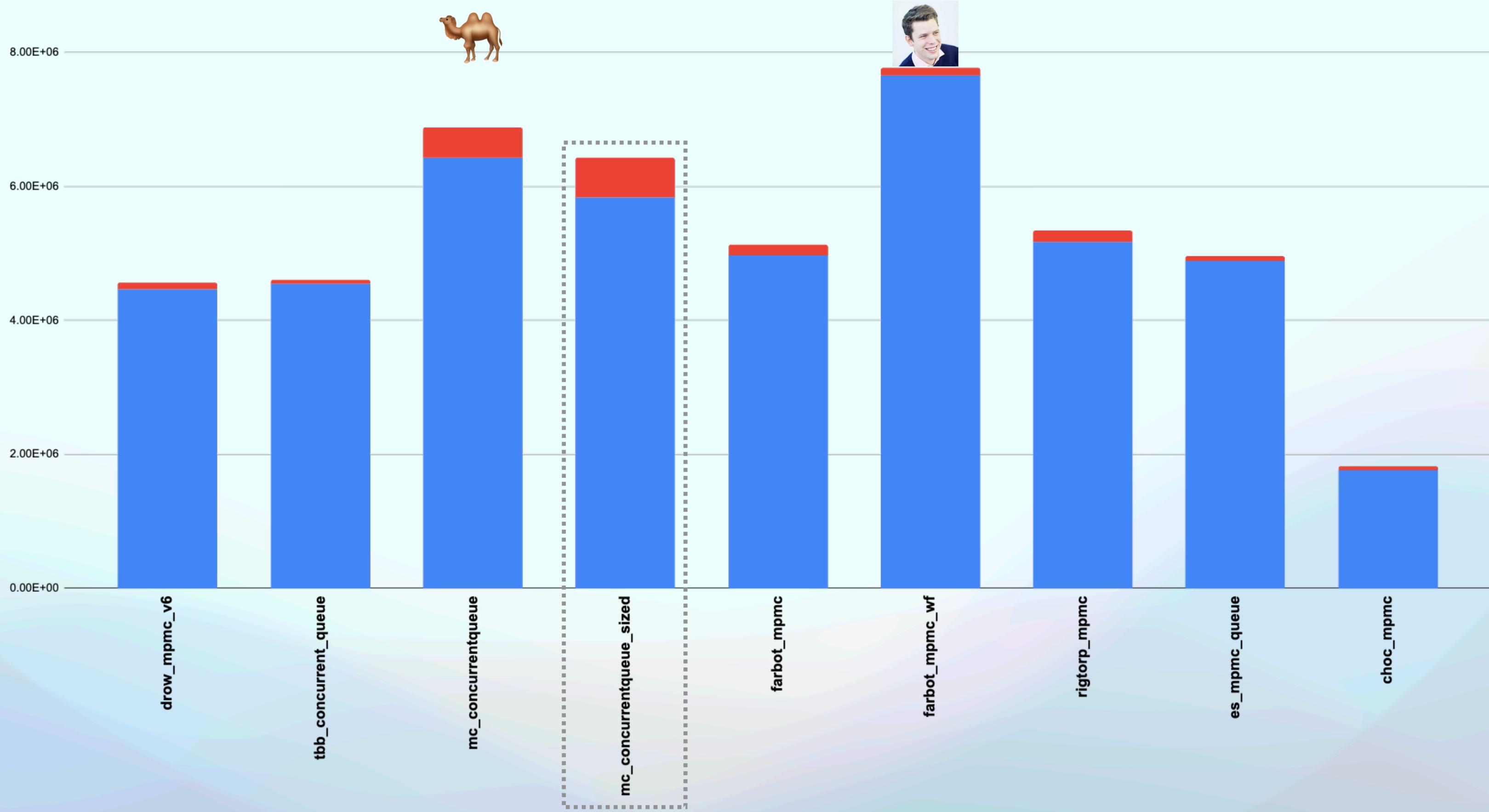
        size.fetch_add (1, std::memory_order_release);
        return true;
    }

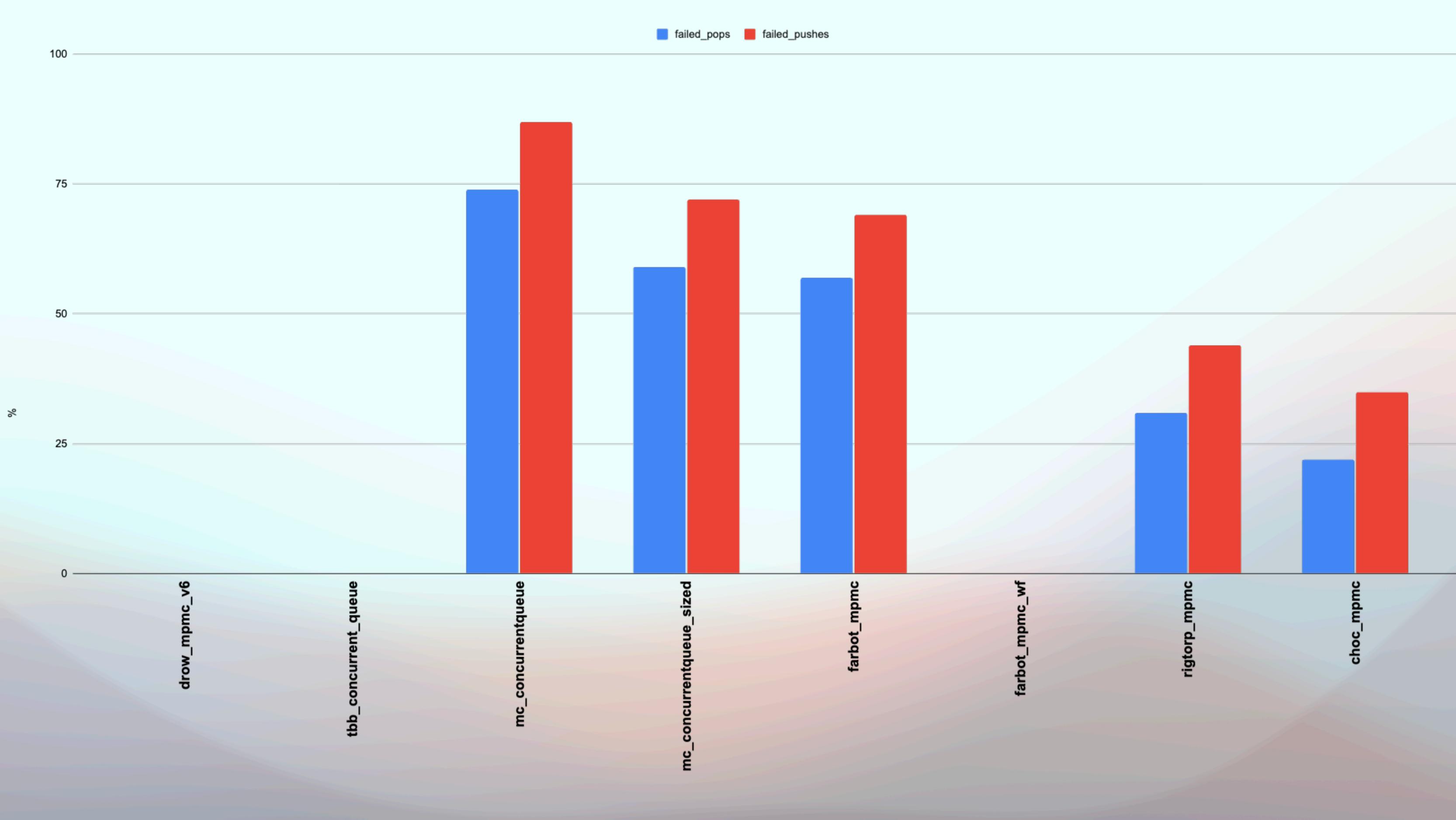
    bool try_pop (T& v)
    {
        if (! queue.try_dequeue (v))
            return false;

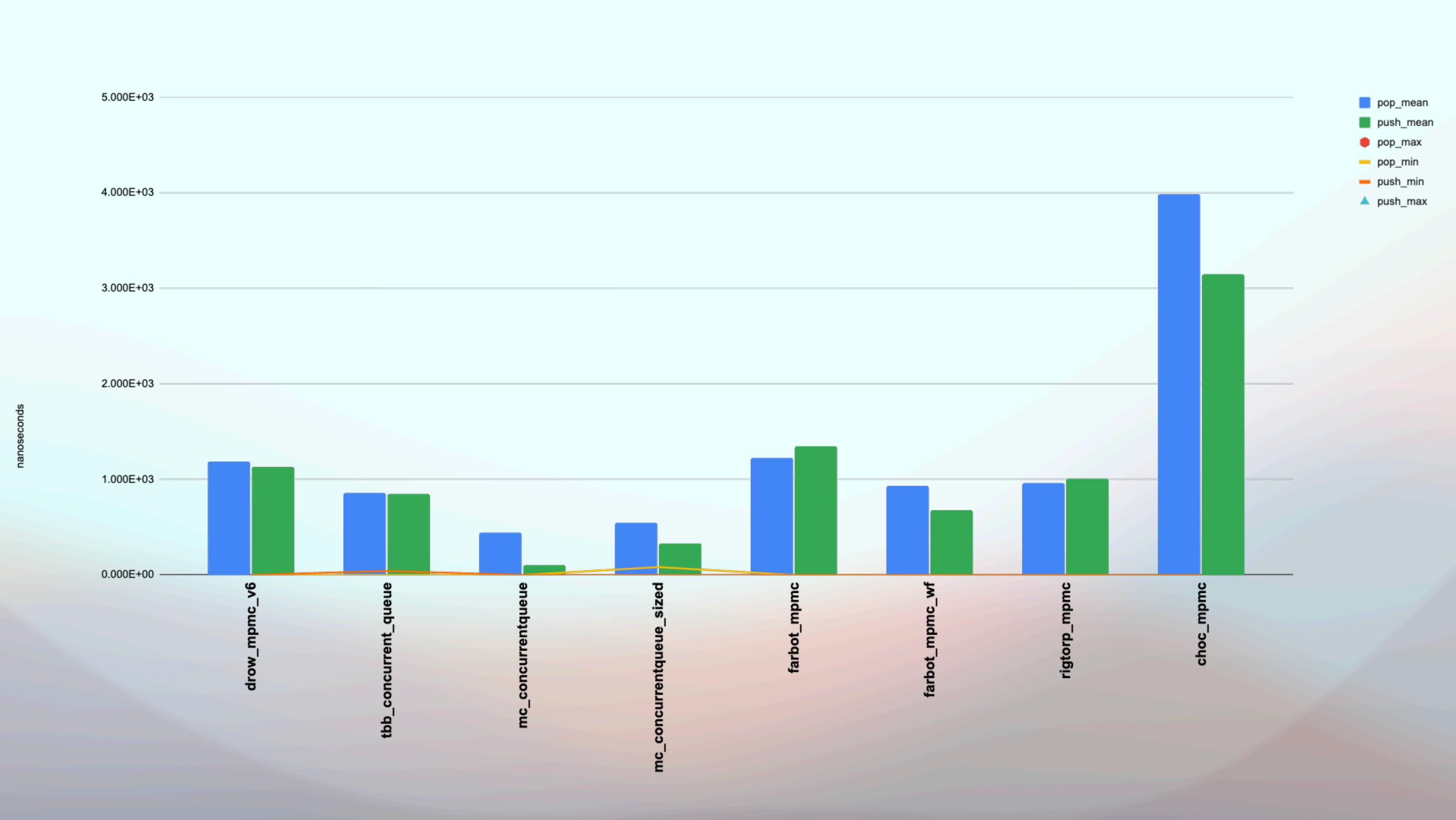
        size.fetch_sub (1, std::memory_order_release);
        return true;
    }

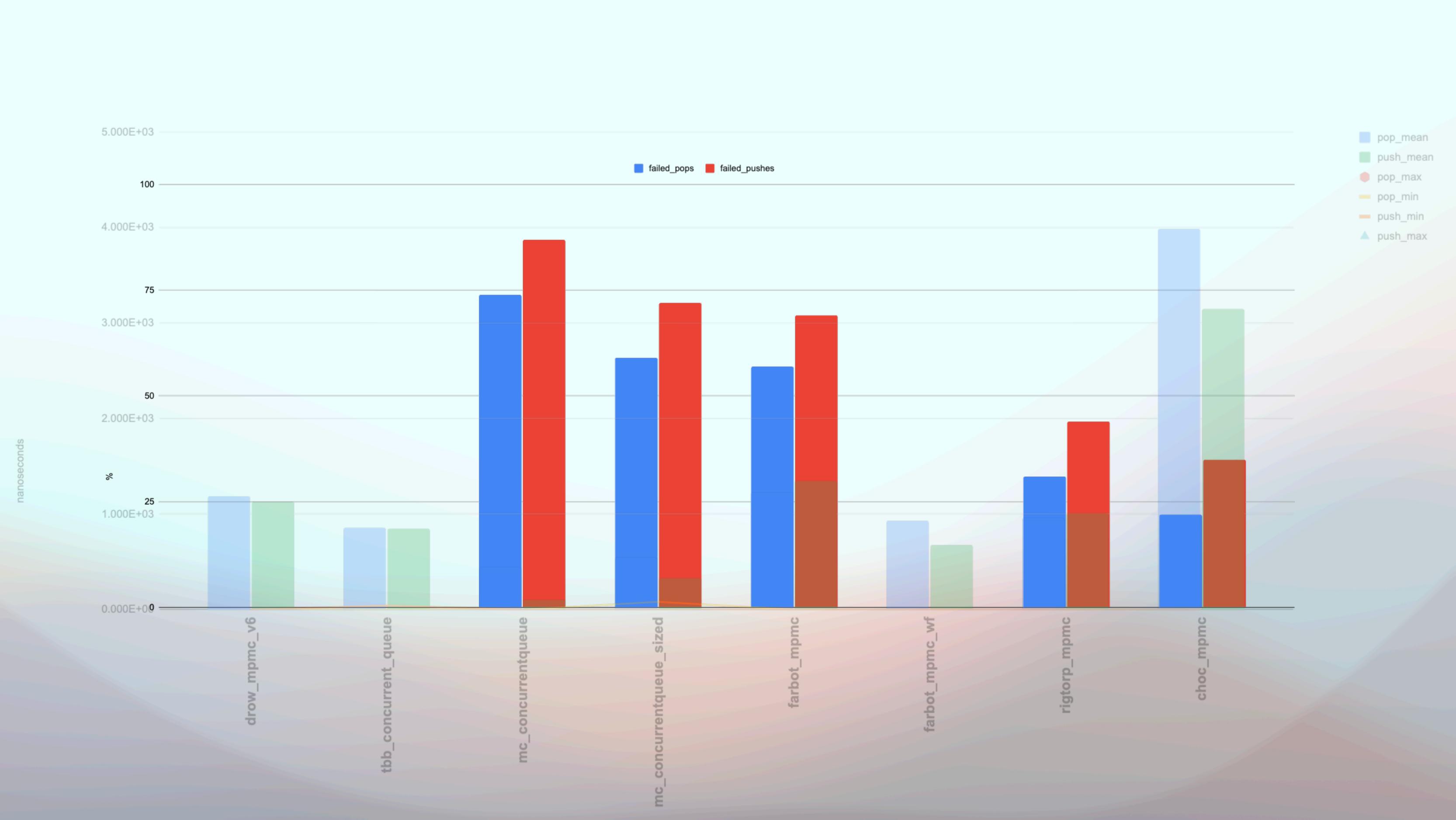
private:
    moodycamel::ConcurrentQueue<T> queue;
    std::atomic<size_t> size { 0 };
};

```









# Additional Optimisations

## CPU Core Pinning

- **CPU Core Pinning**

- Pinning a producer/consumer thread to a CPU core avoids clearing the cache
- Difficult on macOS (mach kernel)

- **Per-thread Caching**

- Cache the read/write indices for each thread
- Allows you to push N cached times without doing any atomic ops

- **Cooperative Consuming**

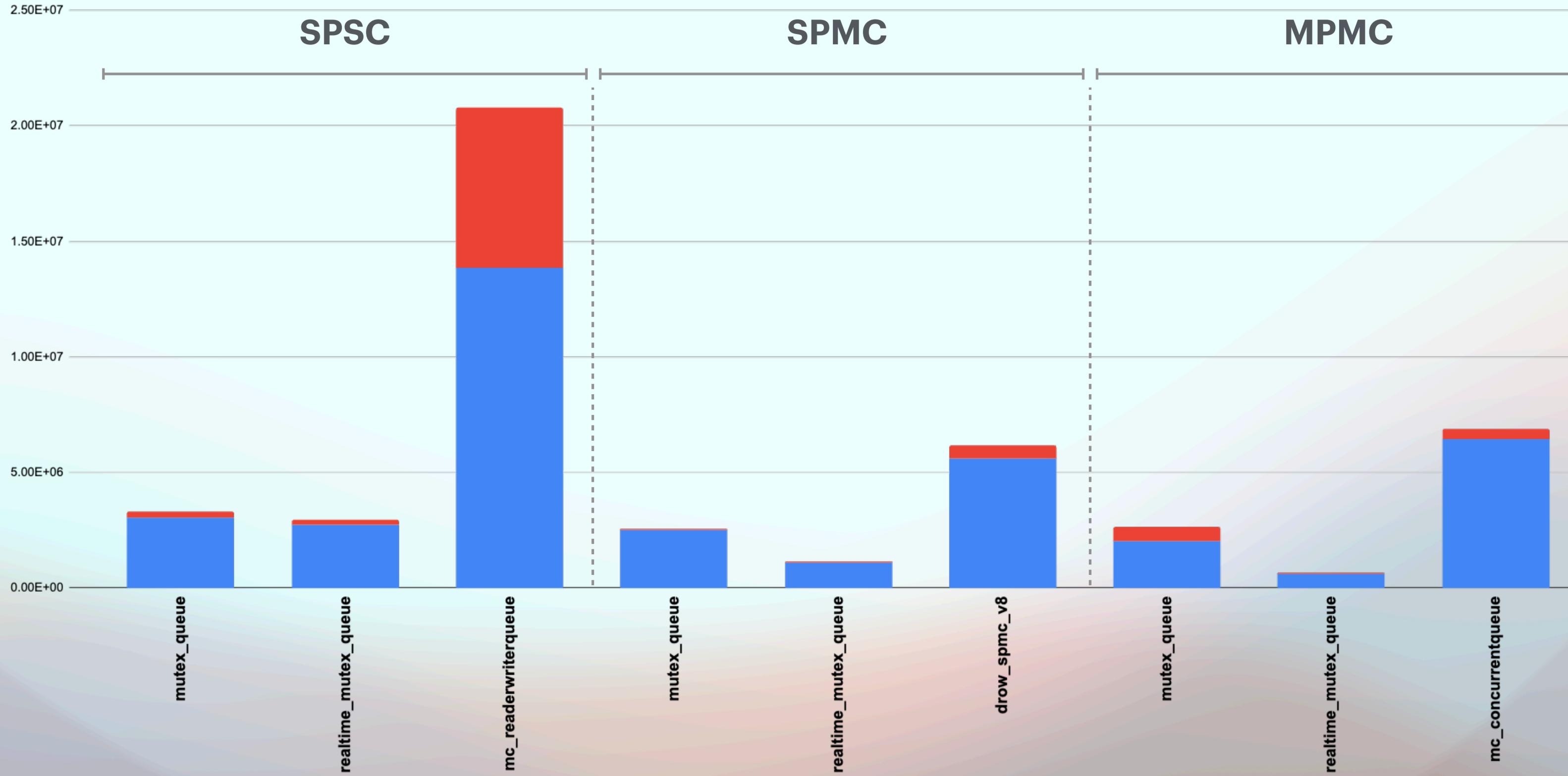
- Producer thread consumes

# Comparison with locks

# 1 SPSC

# 2 SPMC

# 3 MPMC



# Comparison with serial processing

**serial**

**1  
SPSC**

**2  
SPMC**

**3  
MPMC**

2.00E+08

1.50E+08

1.00E+08

5.00E+07

0.00E+00



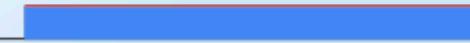
**queue**



**mc\_readerwriterqueue**



**rigtorp\_mpmc**



**mc\_concurrentqueue**

# Comparison between SPMIC/MPPMC



# Trade-offs

Pick the right queue for the job!

- Behaviours
- Throughput
- Latency
- Memory (size)

<b>push</b>	Block on full*	Overwrite when full	Fail/try_
<b>pop</b>	Block on empty*	Return default on empty	Fail/try_
<b>capacity</b>	Static (compile-time)	Fixed (runtime)	Dynamic (grows)
<b>Bulk push/pop</b>	No - single item	Yes - multiple items	
<b>Message size</b>	Fixed	Dynamic	
<b>Gurantees</b>	None - blocking	Lock-free	Wait-free
<b>Message size limit</b>	Limited (8 bytes?)	Unlimted	
<b>Triviality</b>	Trival	Non-trivial	
<b>Num processes</b>	Single	Inter-process	
<b>Threads</b>	Multiple-producers	Multiple-consumers	<i>Max-num threads</i>
<b>Serialisation</b>	Strict global order	Relaxed	



**BONUS**  
LEVEL

# Wait-free MPMC



**Daniel Anderson**

*Assistant Teaching Professor @ CMU*

[danielanderson.net](http://danielanderson.net)

**Carnegie  
Mellon  
University**

Daniel Anderson -- danielanderson.net



**When Lock-Free Still Isn't Enough:**  
*An Introduction to Wait-Free Programming and Concurrency Techniques*

Daniel Anderson



**TO BE  
CONTINUED**

*Please wait for the next level*



# Further reading

- Dave Rowland & Fabian Renn Giles: “Real-time 101”
- Timur Doumler: “Demystifying `std::memory_order`”
- Herb Sutter: “atomic Weapons”
- Christopher Fretz: “Beyond Sequential Consistency”
- Jody Hagins: “Building Robust Inter-Process Queues in C++”
- Erez Strauss: “User API & C++ Implementation of a Multi Producer, Multi Consumer, Lock Free, Atomic Queue”

# Lock-free Queues in the Multiverse of Madness

David Rowland

  @drowaudio

*Questions?*

*Slides/video:*

[drowaudio.github.io/presentations](https://drowaudio.github.io/presentations)

