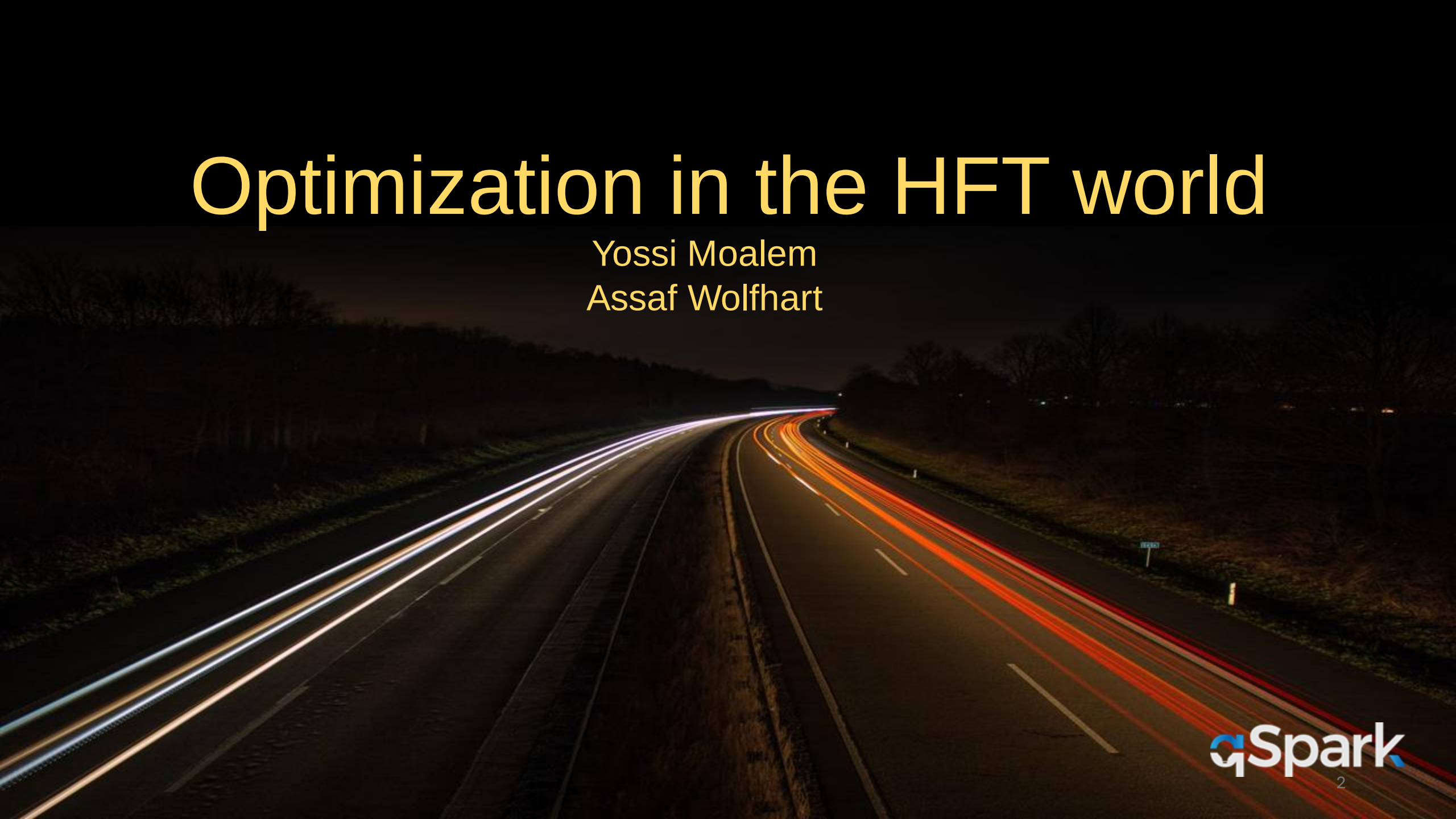

Core C++ 2024
Optimization in the HFT world
Yossi Moalem
Assaf Wolfhart

Optimization in the HFT world

Yossi Moalem
Assaf Wolfhart




Performance
Why?

High level architecture

Real time threads

Admin threads

- Logger
- Monitoring
- Connection management
- Administrative tasks

Avoiding context switches

Divide the threads to 3 tiers:

Real time threads

- Low latency

Admin threads

- High priority

System processes

- Low priority
- Low load

Bypass the scheduler

Consider 8 cores machine, and 5 RT threads

System threads
- Non isolated

Admin threads
- Isolated
- Pin to group

Realtime threads:
- Isolated
- Pin thread to a dedicated core
- Spin

Admin Threads / system threads
- Non isolated

Networking kernel bypass

Kernel	• ~4000 ns
OnLoad	•Drop-in replacement •~300 - 450 ns (UDP/TCP)
TCP Direct	•Proprietary API • ~15 - 22ns(UDP/TCP)
EF_VI	• Low level API • Direct access to SF NIC • Direct access to the NIC queues

Message stream

Only the rare case

Optimizing for the rare case

	Base
Nontrade event latency	2 µs
Trade event latency	5 µs
Total time *	2005 µs

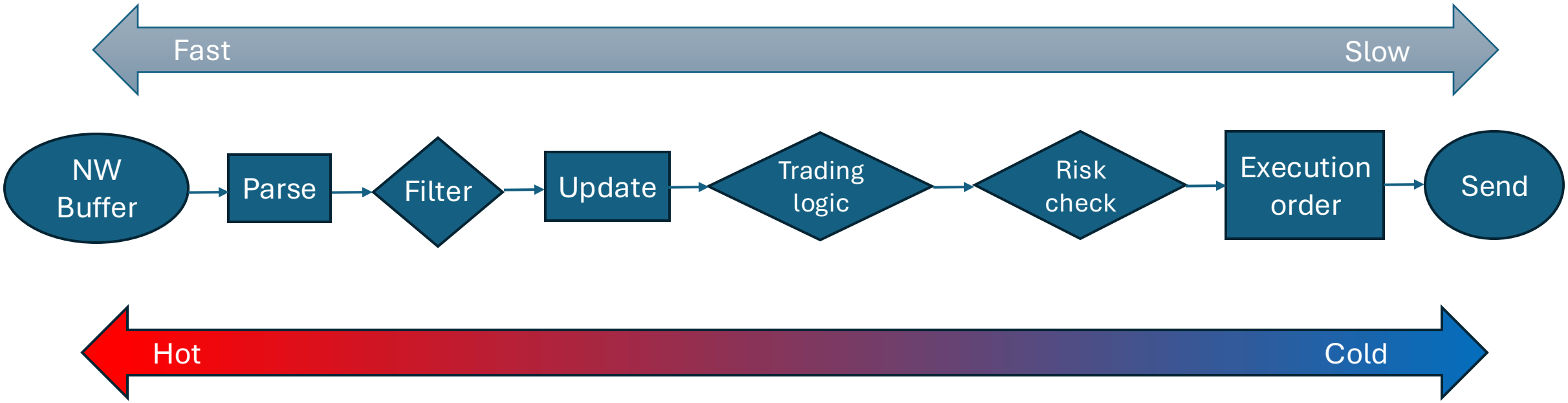
* Assume 1000 non-trading events and 1 trading event:

Optimizing the rare case

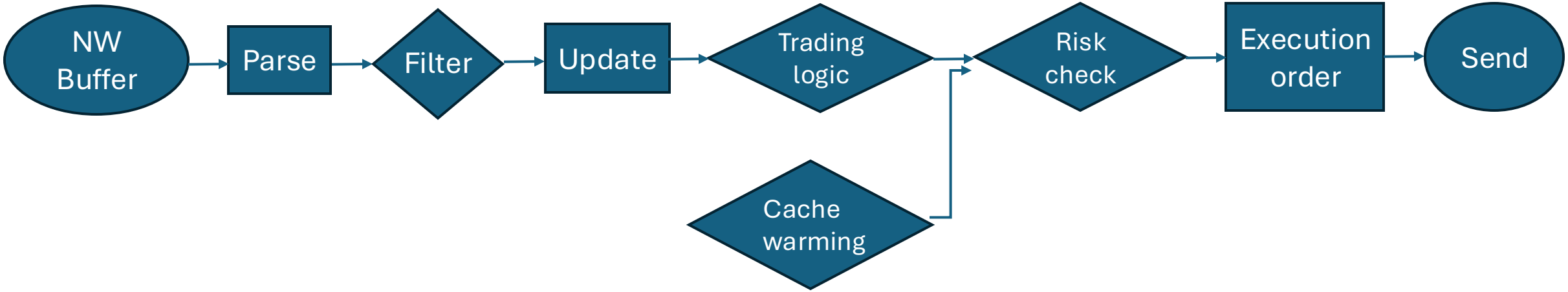
	Base	Optimization?
Nontrade event latency	2 µs	3 µs
Trade event latency	5 µs	4 µs
Total time *	2005 µs	3004 µs

* Assume 1000 non-trading events and 1 trading event:

The most important optimization



The most important optimization



Cache warm should not be aware that it is cache warming

Before any cache warming calls:

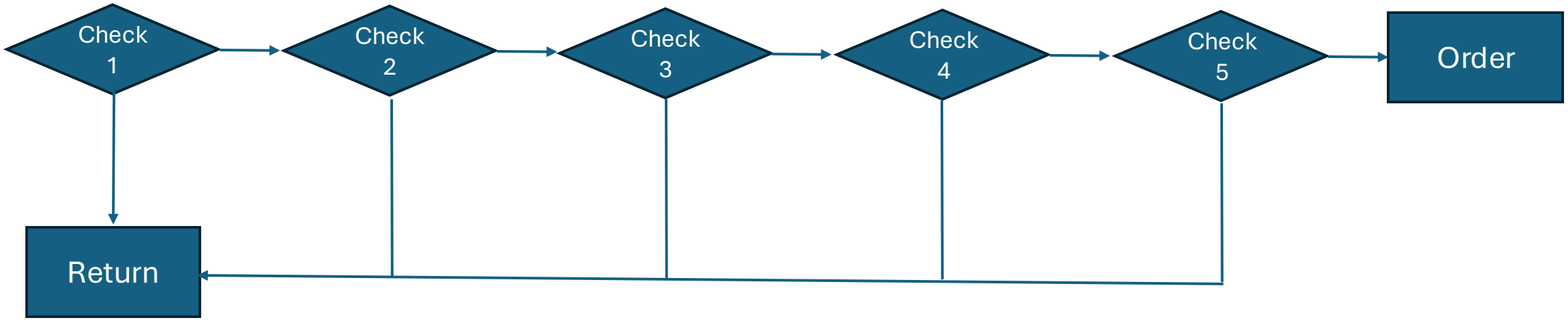
```
++totalNumberOfOrders;
```

Adapted to cache warming:

```
totalNumberOfOrders += ( not isCacheWarming );
```

```
++totalNumberOfOrders[ isCacheWarming ];
```

Risk checking



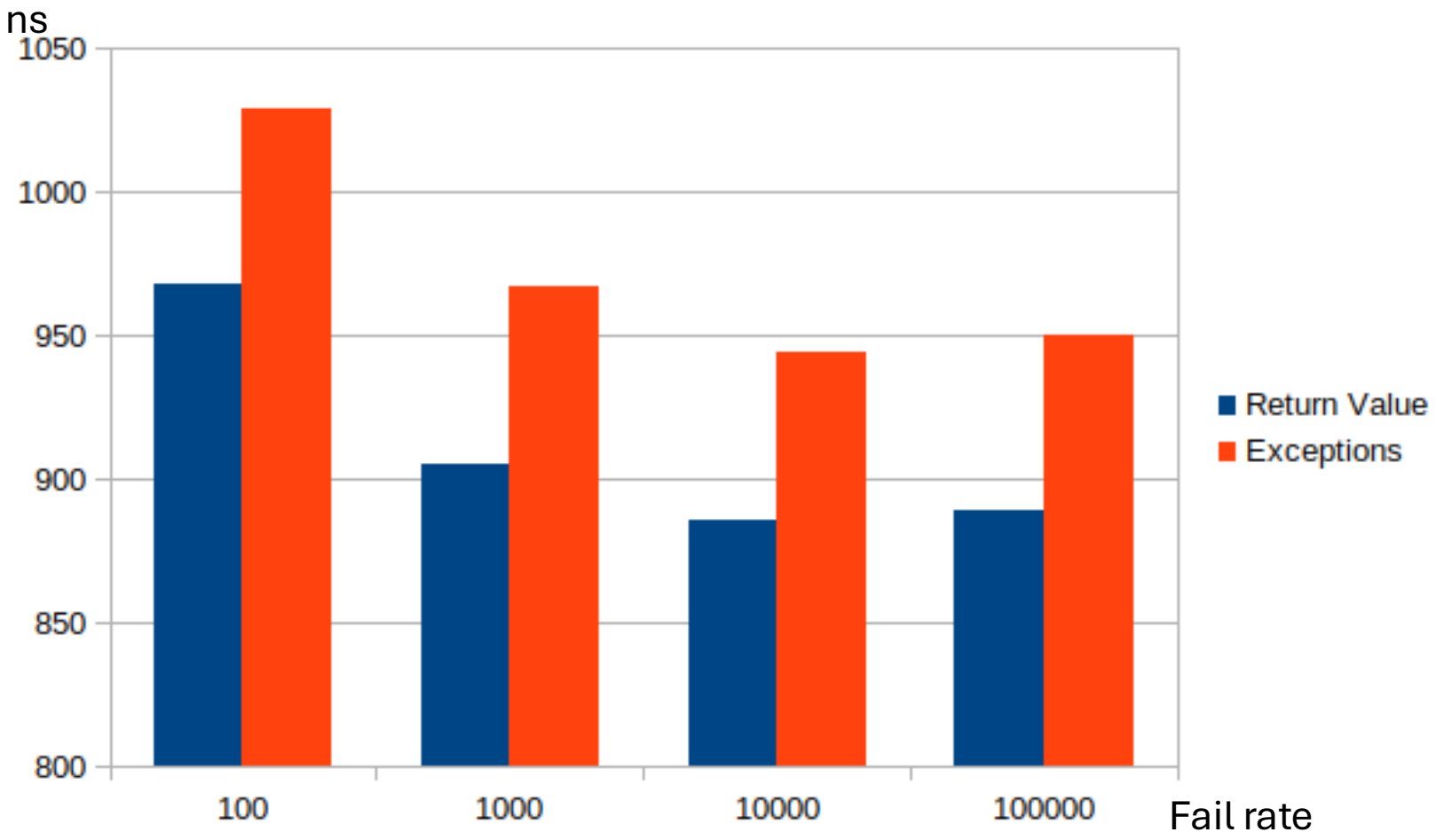
Failure handling

```
if (! check1( ) )
    { return -1; }
if (! check2( ) )
    { return -1; }
if (! check3( ) )
    { return -1; }
if (! check4( ) )
    { return -1; }
if (! check5( ) )
    { return -1; }
if (! check6( ) )
    { return -1; }
if (! check7( ) )
    { return -1; }
if (! check8( ) )
    { return -1; }
if (! check9( ) )
    { return -1; }
if (! check10( ) )
    { return -1; }
return 0;
```

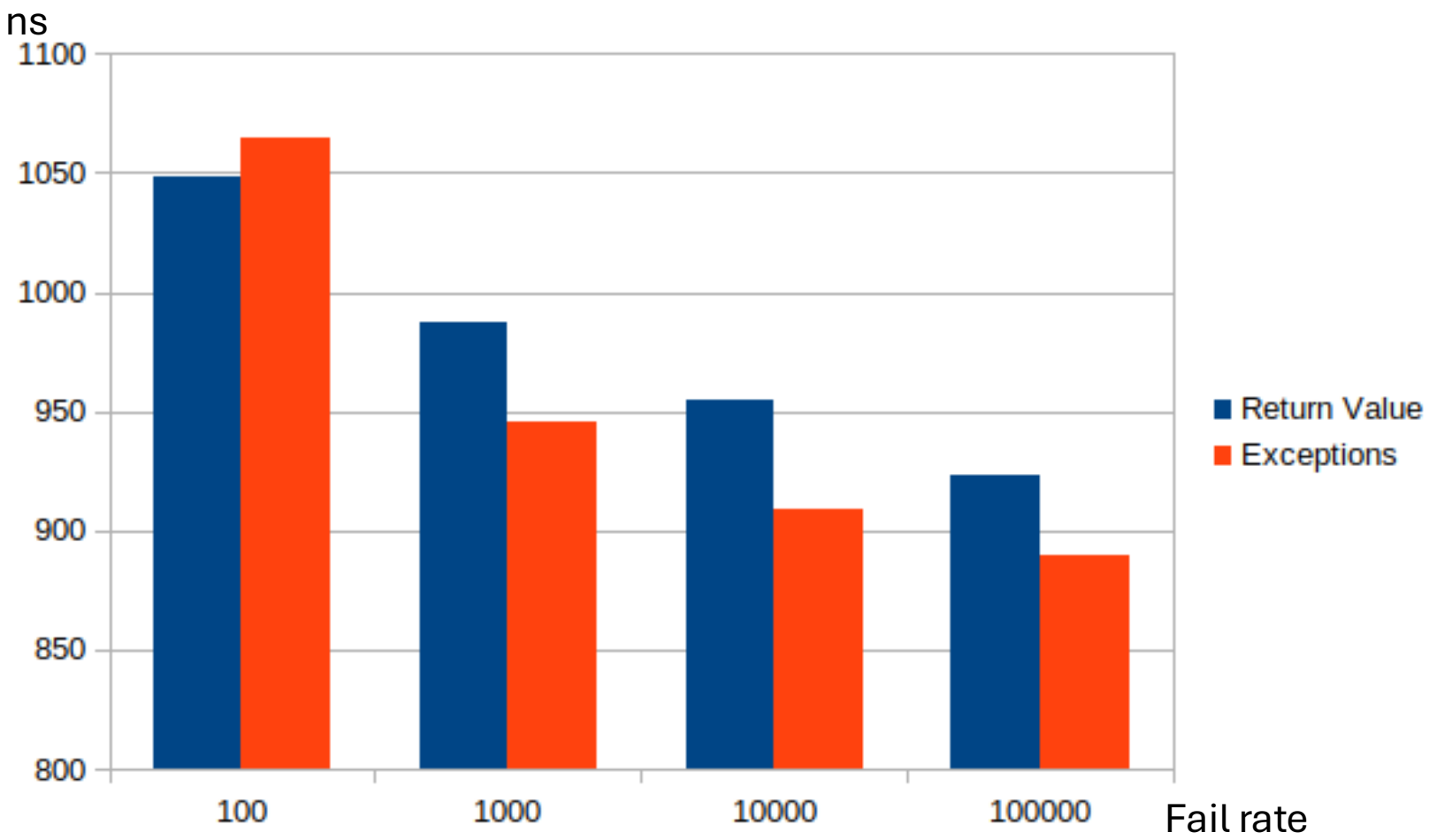
```
try{
    check1( );
    check2( );
    check3( );
    check4( );
    check5( );
    check6( );
    check7( );
    check8( );
    check9( );
    check10( );
    return 0;
}
catch (...){
    return -1;
}
```

Nothing is an optimization unless measured

Failure handling



Failure handling



Micro-benchmark is cool, But…

- Make sure you measure the correct thing
- Make sense of the results
- Always measure your app, and in a real scenario

Sharing data between threads

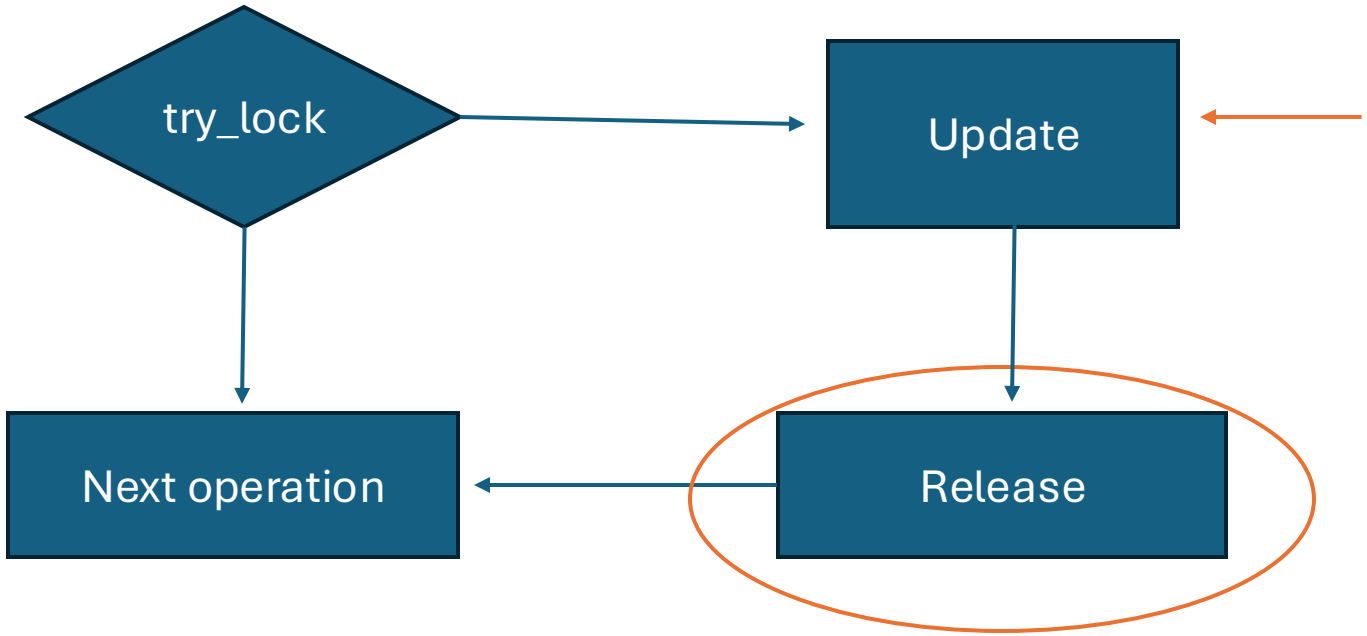
```
std::atomic<MyDataType> sharedData;
```

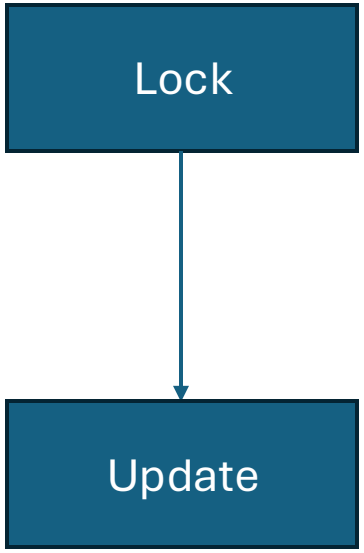
```
static_assert (std:atomic<MyDataType>::is_always_lock_free);
```

Let’s try to find relaxations:

- Whole structure must be atomic
- Can we fail the update

Fail-able update





Spinlock to the rescue

```
struct spinLock {
   std::atomic<bool> lock_ = {false};

   void lock() {
     while(lock_.exchange(true, std::memory_order_acquire));
   }

   void unlock() {
     lock_.store(false, std::memory_order_release);
   }
};
```

Better spinlock

```
void lock() {
    for (;;) {
        if (!lock_.exchange(true,std::memory_order_acquire)) {
            break;
        }
        while (lock_.load(std::memory_order_relaxed)) {
            considerYield();
        }
    }
}
```

Sharing data between threads

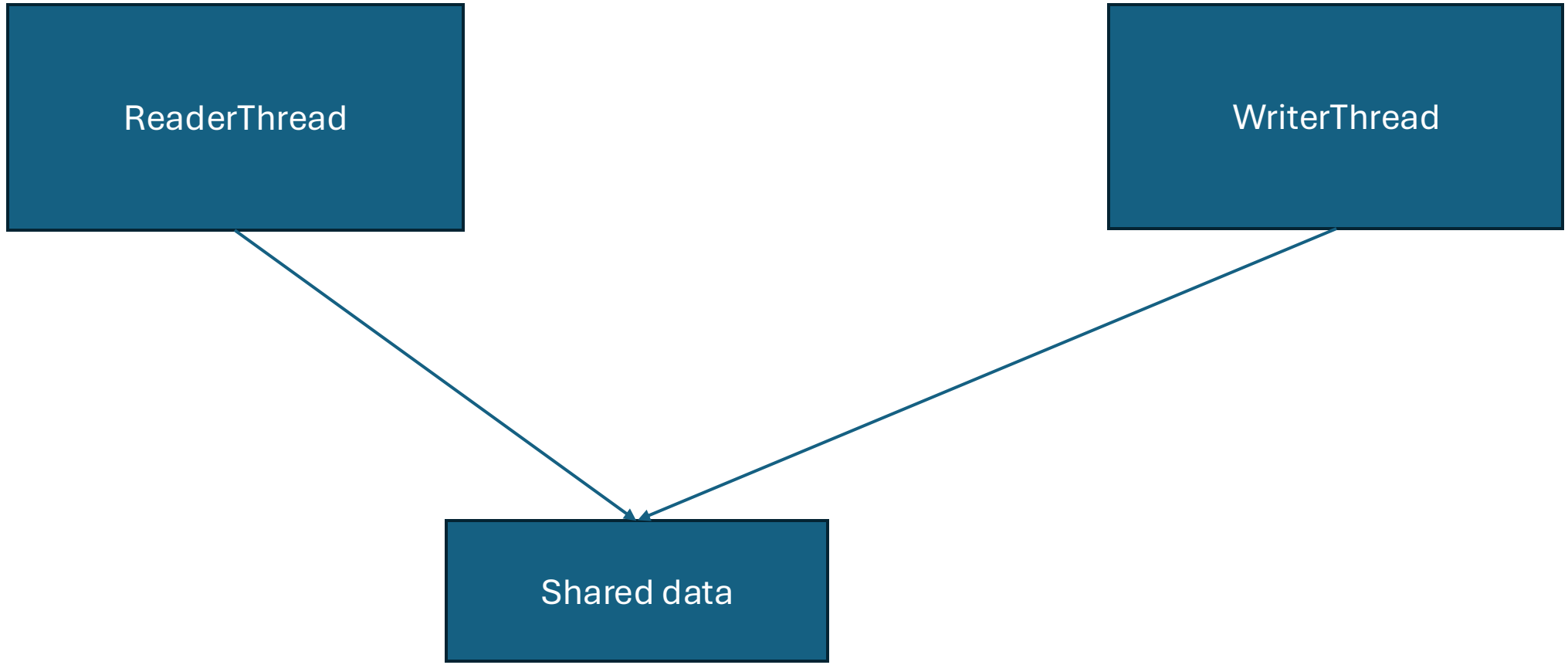
```
std::atomic<MyDataType> sharedData;
```

```
static_assert (std:atomic<MyDataType>::is_always_lock_free);
```

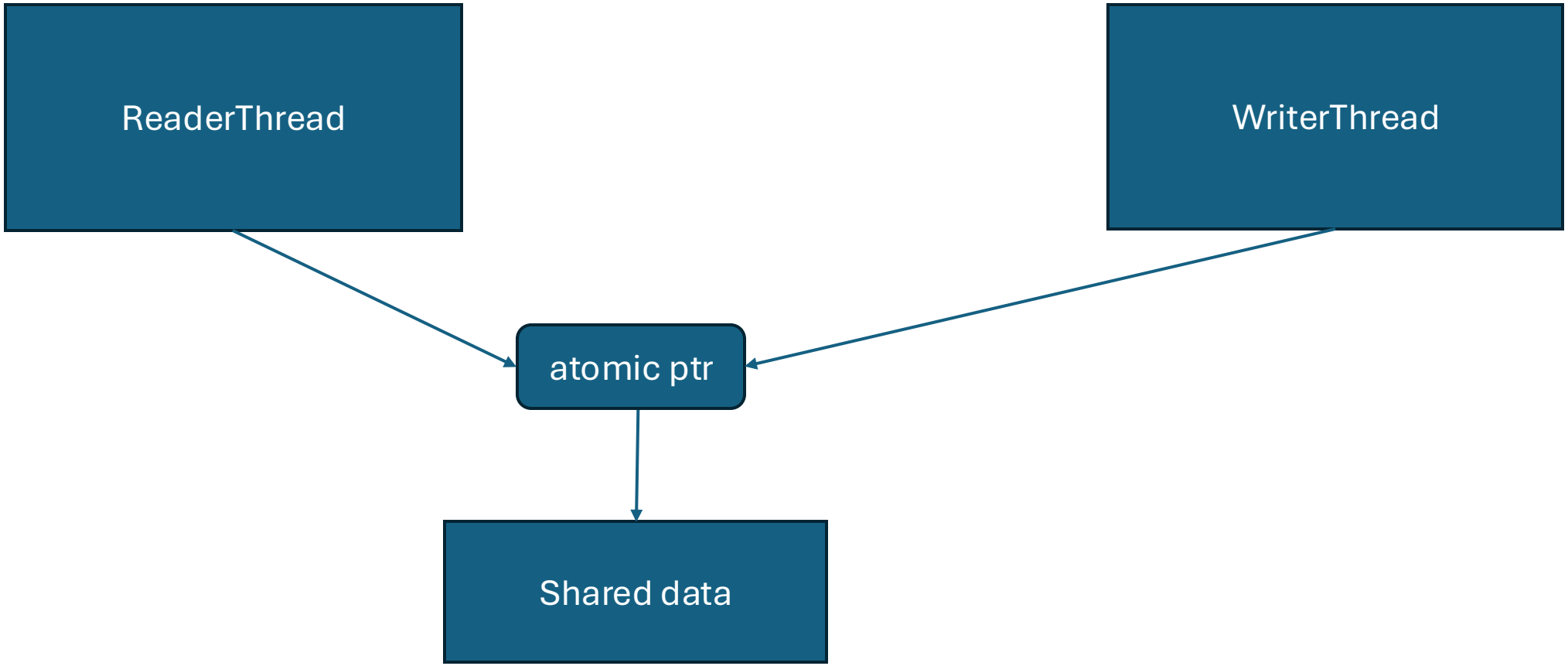
Let’s try to find relaxations:

- Whole structure must be atomic
- Can we fail the update
- Are updates dependent upon each other
- Number of readers/writers
- Realtime thread is reader or writer

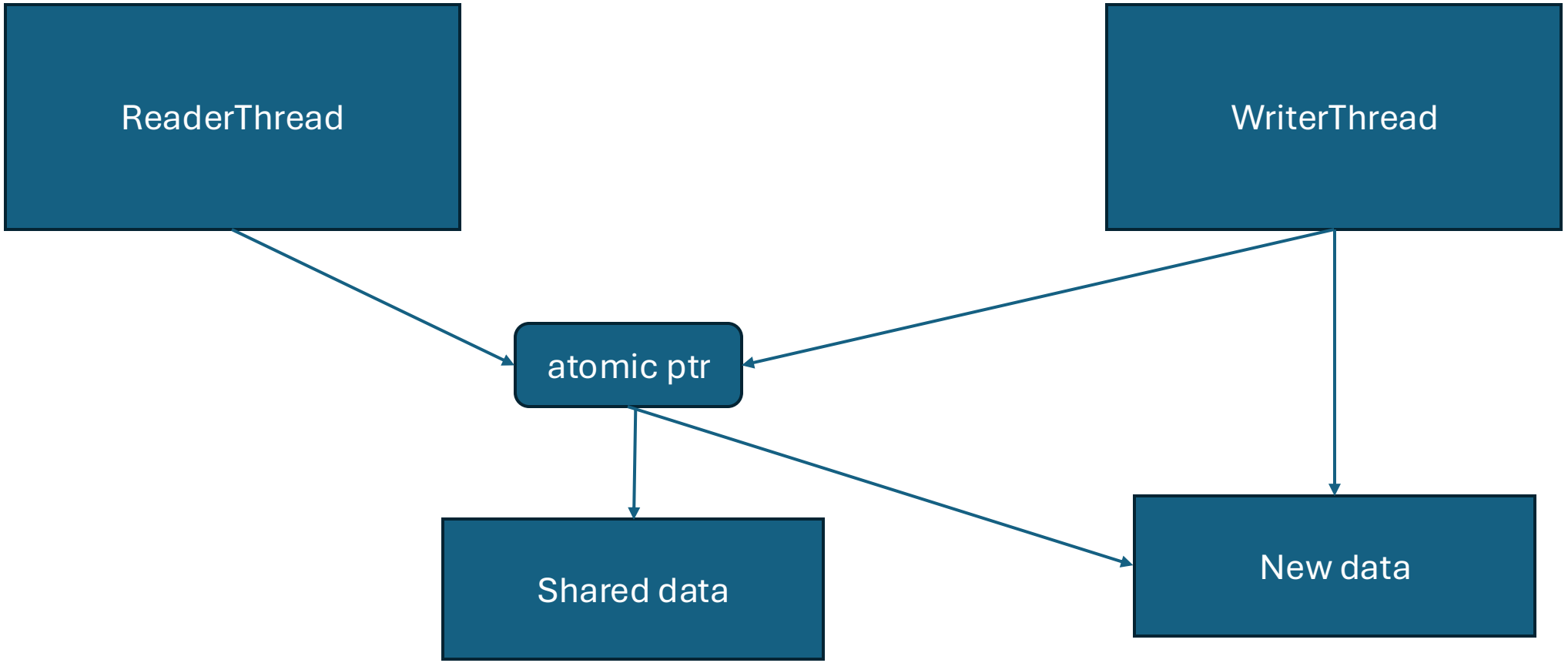
Updating shared data



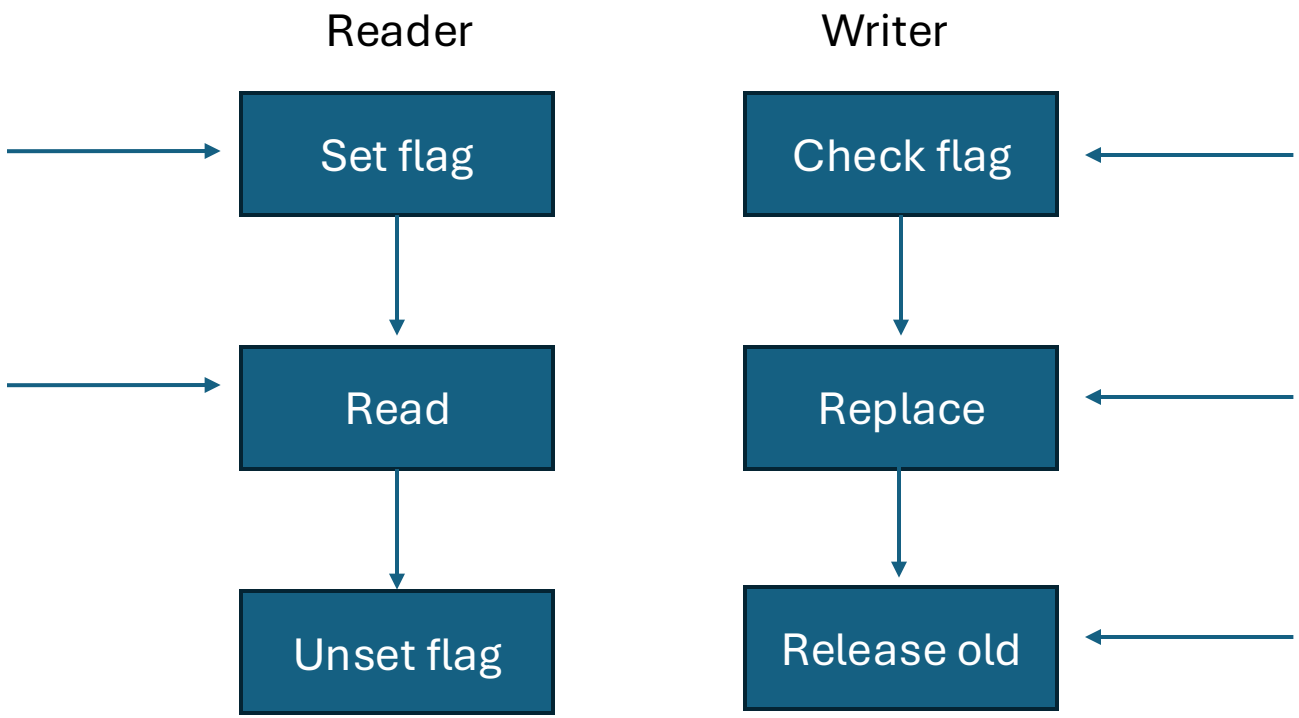
Updating shared data



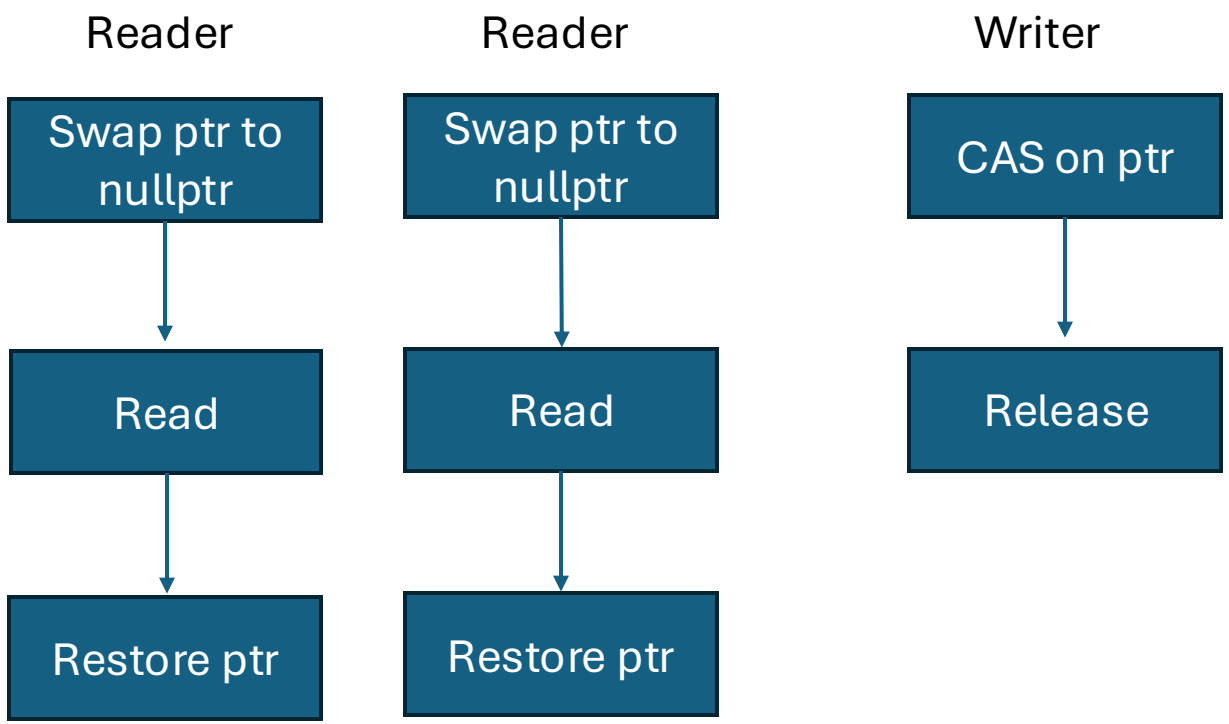
Updating shared data



Who does the cleanup: Naive approach



Who does the cleanup: CAS loop



Pre-mature generalization is the root of all evil

- Take advantage of any relaxation you have
- Use the most specific data structure

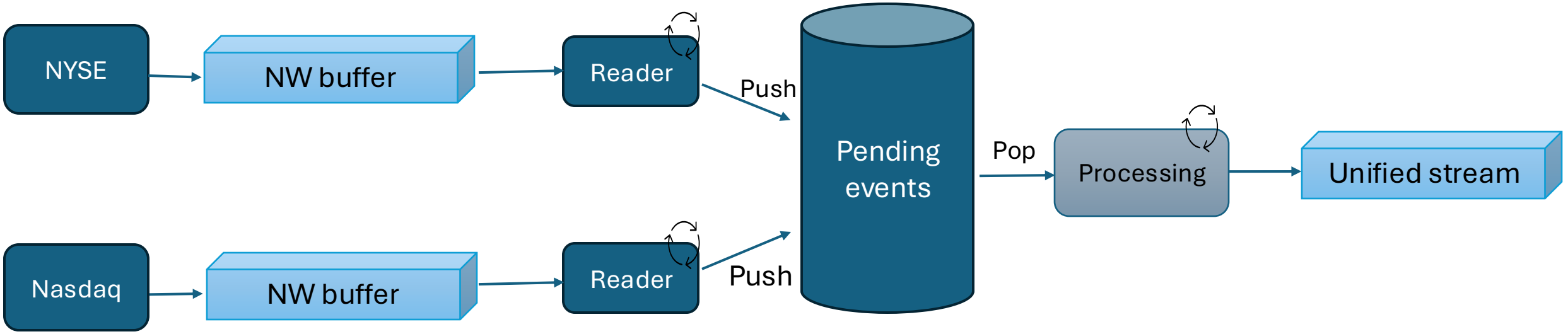
Optimizing for throughput

Broad market view

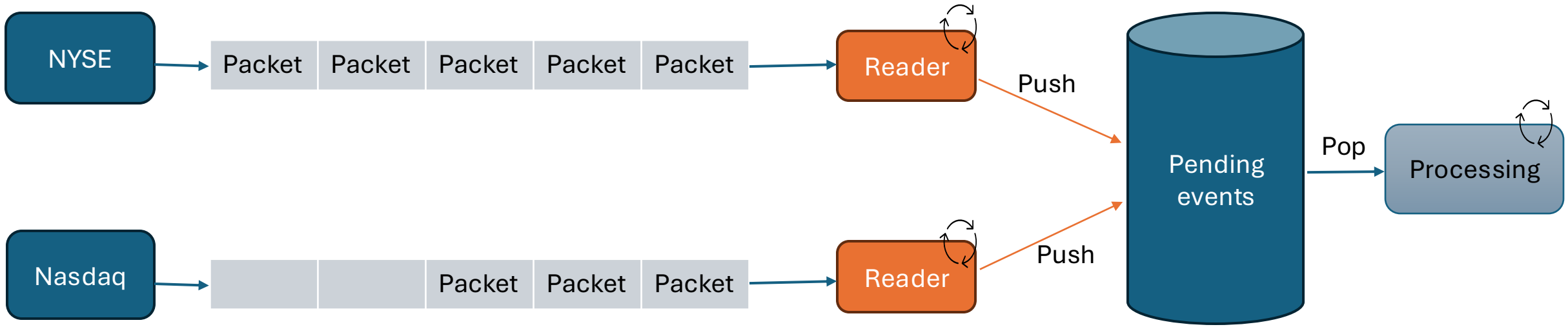
Latency is still important

But second to throughput

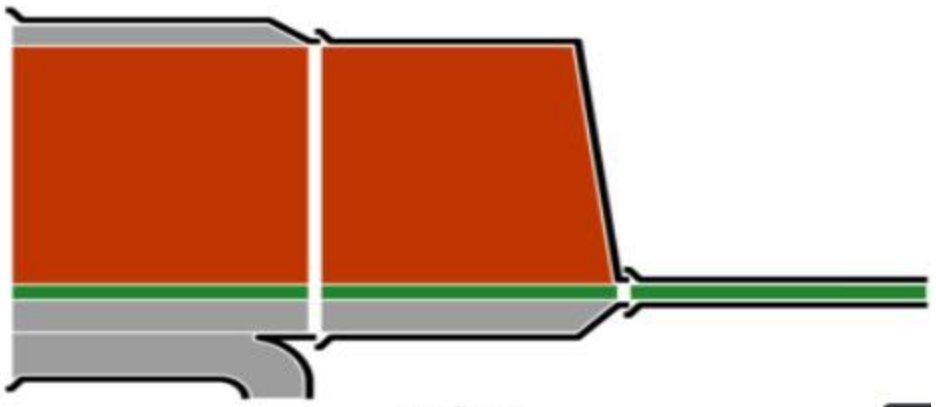
High level architecture



Drops still happen

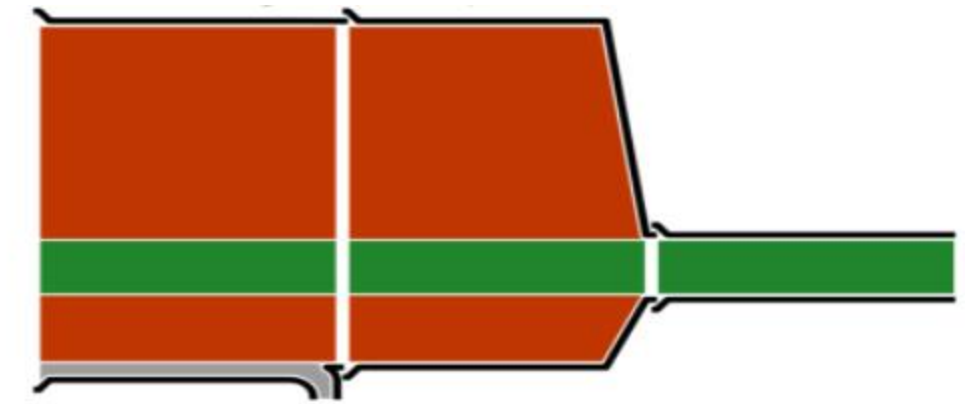


vTune push and pop pipe graphs (during the open)



µPipe

Retiring:	4.4%	of Pipeline Slots
Front-End Bound:	6.0%	of Pipeline Slots
Bad Speculation:	13.2%	of Pipeline Slots
Back-End Bound:	76.4%	of Pipeline Slots
Memory Bound:	67.3%	of Pipeline Slots
L1 Bound:	63.9%	of Clockticks
DTLB Overhead:	13.9%	of Clockticks
Loads Blocked by Store Forwarding:	9.3%	of Clockticks
Lock Latency:	22.3%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.2%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	3.2%	of Clockticks
DRAM Bound:	44.7%	of Clockticks



µPipe

Retiring:	15.5%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	4.9%	of Pipeline Slots
Back-End Bound:	79.5%	of Pipeline Slots
Memory Bound:	60.4%	of Pipeline Slots
L1 Bound:	55.9%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.0%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	19.1%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.7%	of Clockticks
Cycles of 0 Ports Utilized:	54.1%	of Clockticks
Cycles of 1 Port Utilized:	19.5%	of Clockticks
Cycles of 2 Ports Utilized:	16.5%	of Clockticks
Cycles of 3+ Ports Utilized:	10.0%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	

Lock free queues types

Consumers

- Single
- Multi

Producers

- Single
- Multi

Pop on empty

- Retrurn false
- Return sentinel

Push when full

- Return false
- Overwrite

Favour

- Readers
- Writers

Can we do better?

Key design concepts

Single consumer multi producer

Producers count is known at compile time

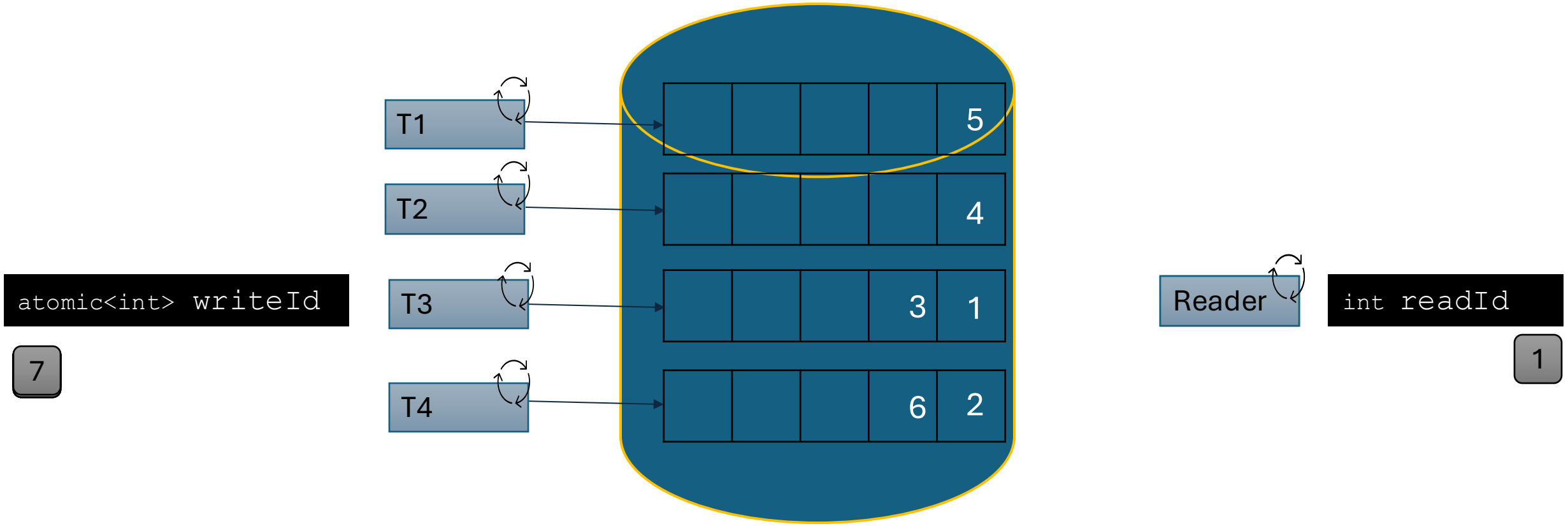
Favour writers

Reduce writers sharing

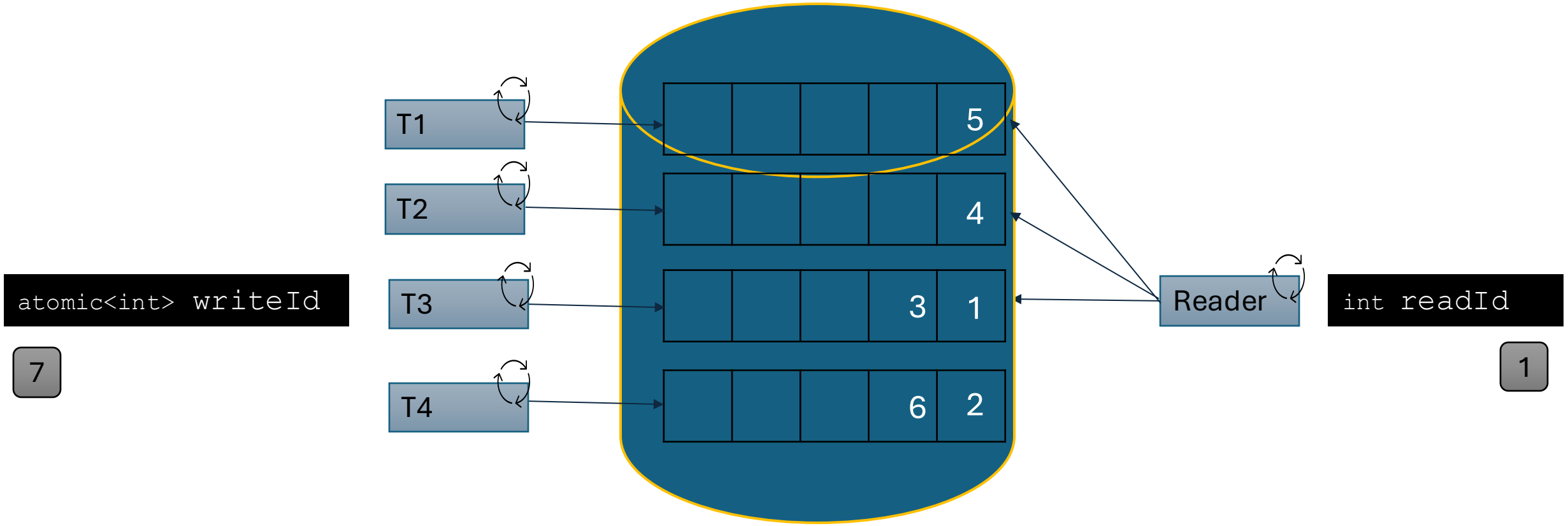
Reduce reader/writer sharing

Improve memory ordering

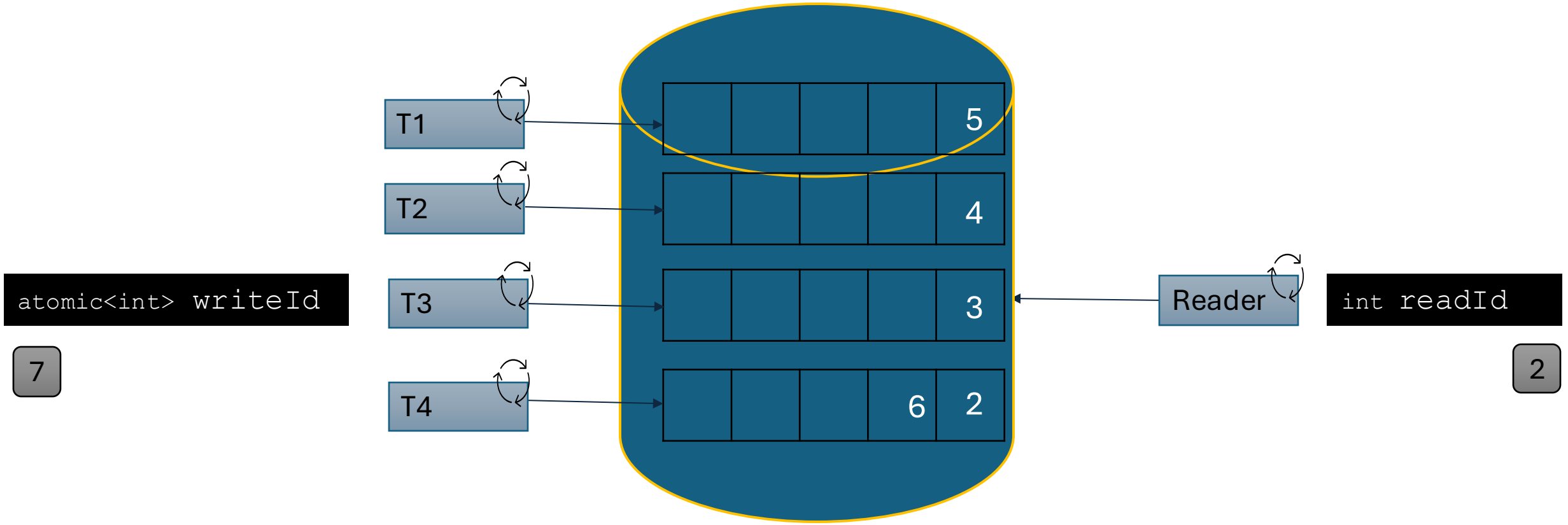
High level design



High level design



High level design



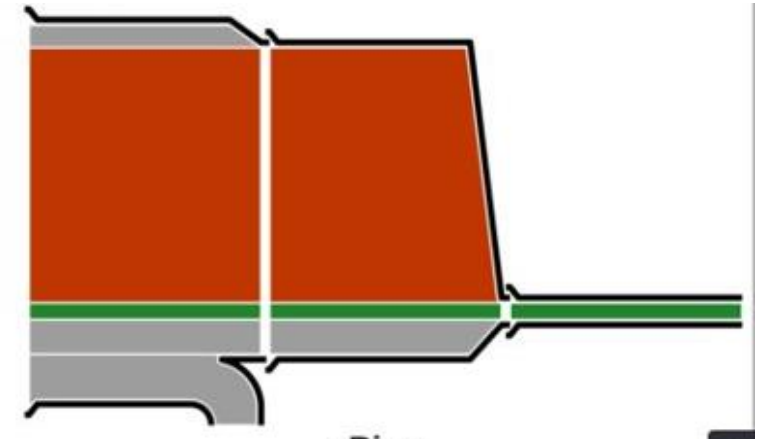
Push

```
bool SCLFQ::push(U&& item){
  const thread_local size_t queueIndex = getQueueIndex();
  return m_cyclicQueues[queueIndex].push(std::forward<U>(item), m_writeId);
}
```

```
bool InternalCyclicQueue::push(U&& item, std::atomic<int_fast64_t>& writeId){
  if (hasSpaceLeft())   {
      Cell& cell = m_queue[m_writerIndex];
      cell.item = std::forward<U>(item);
      markAsConsumable(cell, writeId);
      incrementSubQueueIndex(m_writerIndex);
      return true;
   }
   Return false;
}
```

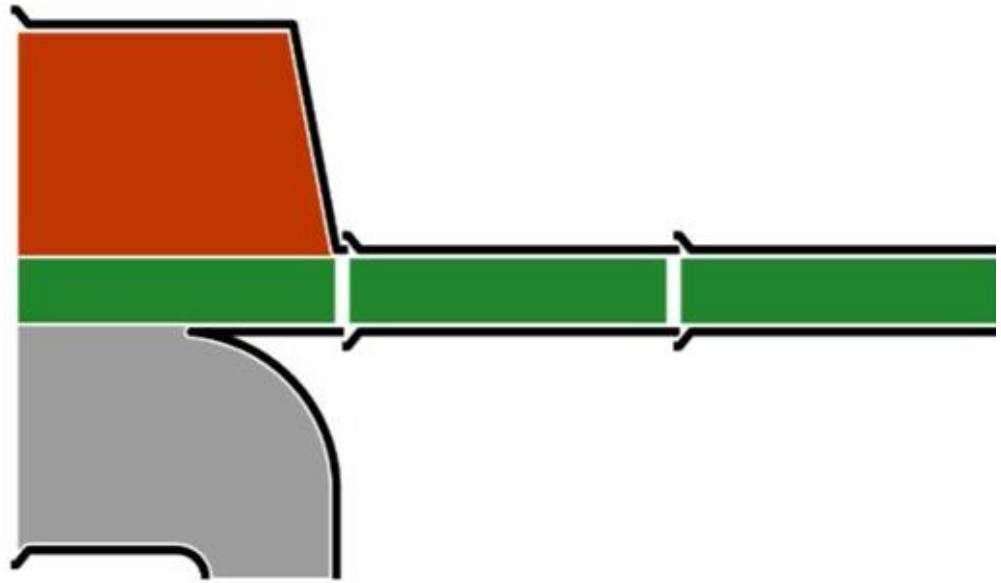
```
void InternalCyclicQueue::markAsConsumable(Cell& cell, std::atomic<int_fast64_t>&
writeId) {
  cell.id.store(writeId.fetch_add(1, std::memory_order_relaxed),
std::memory_order_release);
}
```

Push



µPipe

Retiring:	4.4%	of Pipeline Slots
Front-End Bound:	6.0%	of Pipeline Slots
Bad Speculation:	13.2%	of Pipeline Slots
Back-End Bound:	76.4%	of Pipeline Slots
Memory Bound:	67.3%	of Pipeline Slots
L1 Bound:	63.9%	of Clockticks
DTLB Overhead:	13.9%	of Clockticks
Loads Blocked by Store Forwarding:	9.3%	of Clockticks
Lock Latency:	22.3%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.2%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	3.2%	of Clockticks
DRAM Bound:	44.7%	of Clockticks



µPipe

Retiring:	13.9%	of Pipeline Slots
Front-End Bound:	46.3%	of Pipeline Slots
Front-End Latency:	18.5%	of Pipeline Slots
Front-End Bandwidth:	27.8%	of Pipeline Slots
Bad Speculation:	46.3%	of Pipeline Slots
Back-End Bound:	0.0%	of Pipeline Slots

Pop

```
bool SCLFQ::pop(Item& item){
    for (uint_fast32_t i = 0; i < ProducersCount; ++i)
        if (m_cyclicQueues[i].topId() == m_readId){
            m_cyclicQueues[i].pop(item);
            ++m_readId;
            return true;
        }
    return false;
}
```

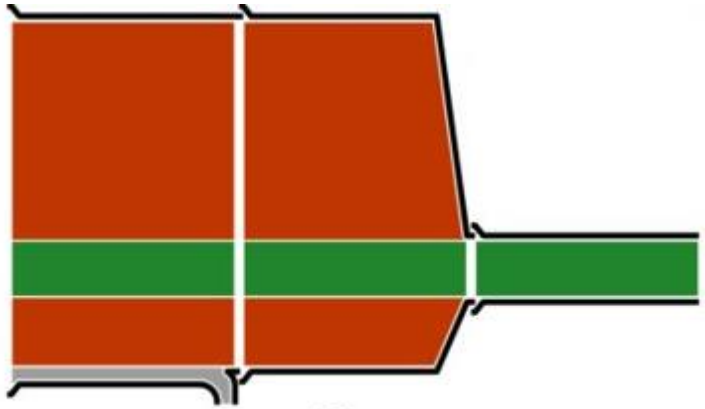
```
[nodiscard]] int_fast64_t InternalCyclicQueue::topId() const {
  return m_queue[m_readerIndex].id.load(std::memory_order_acquire);
}
```

```
void InternalCyclicQueue::pop(T& item){
  item = std::move(cell.item);
  cell.id.store(RESERVED_FOR_WRITING, std::memory_order_release);
  incrementSubQueueIndex(m_readerIndex);
}
```

Branchless pop

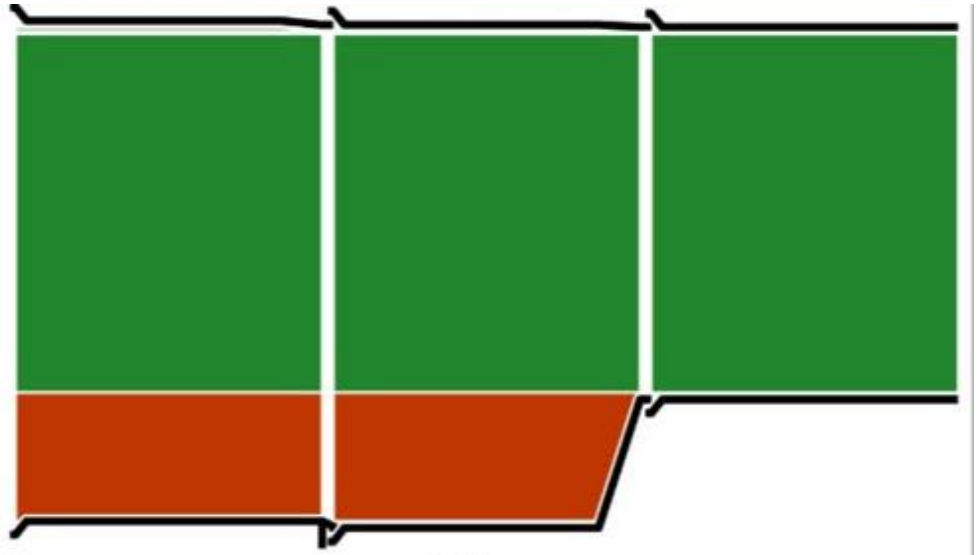
```
bool SCLFQ::popBranchless(Item& item) {    //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i - 1].topId() == m_readId);

    if (idx == 0)
        return false;

    m_cyclicQueues[idx - 1].pop(item);
    ++m_readId;
    return true;
}
```

Pop



µPipe

Retiring:	15.5%	of Pipeline Slots
Front-End Bound:	0.1%	of Pipeline Slots
Bad Speculation:	4.9%	of Pipeline Slots
Back-End Bound:	79.5%	of Pipeline Slots
Memory Bound:	60.4%	of Pipeline Slots
L1 Bound:	55.9%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.0%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	19.1%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	17.7%	of Clockticks
Cycles of 0 Ports Utilized:	54.1%	of Clockticks
Cycles of 1 Port Utilized:	19.5%	of Clockticks
Cycles of 2 Ports Utilized:	16.5%	of Clockticks
Cycles of 3+ Ports Utilized:	10.0%	of Clockticks
Vector Capacity Usage (FPU):	0.0%	



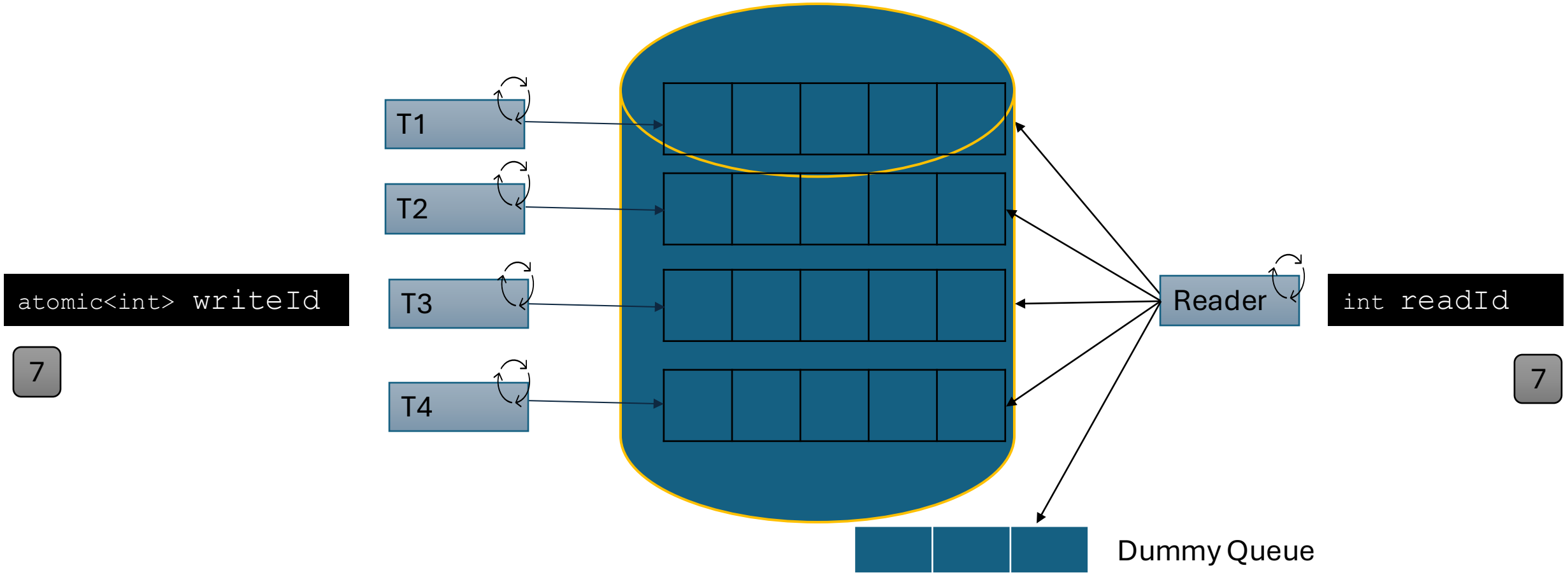
µPipe

Retiring:	73.7%	of Pipeline Slots
Light Operations:	73.7%	of Pipeline Slots
Heavy Operations:	0.0%	of Pipeline Slots
Front-End Bound:	0.9%	of Pipeline Slots
Bad Speculation:	0.0%	of Pipeline Slots
Back-End Bound:	26.9%	of Pipeline Slots
Memory Bound:	0.5%	of Pipeline Slots
Core Bound:	26.4%	of Pipeline Slots
Divider:	0.0%	of Clockticks

But we still have one more branch

```
bool SCLFQ::popBranchless(Item& item) {    //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i - 1].topId() == m_readId);

    if (idx == 0)
        return false;

    m_cyclicQueues[idx - 1].pop(item);
    ++m_readId;
    return true;
}
```

Removing the last branch



Removing the last branch: Required changes:

Install Dummy queue at `m_cyclicQueues[0]`

```
bool SCLFQ::popBranchless(Item& item) {    //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i - 1].topId() == m_readId);

    if (idx == 0)
        return false;

    m_cyclicQueues[idx - 1].pop(item);
    ++m_readId;
    return true;
}
```

Removing the last branch

Install Dummy queue at `m_cyclicQueues[0]`

```
bool SCLFQ::popBranchless(Item& item) {    //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i - 1].topId() == m_readId);

    m_cyclicQueues[idx - 1].pop(item);
    ++m_readId;
    return true;
}
```

Removing the last branch

Install Dummy queue at `m_cyclicQueues[0]`

```
bool SCLFQ::popBranchless(Item& item) {    //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i].topId() == m_readId);

    m_cyclicQueues[idx].pop(item);
    ++m_readId;
    return true;
}
```

Removing the last branch

Install Dummy queue at `m_cyclicQueues[0]`

```
bool SCLFQ::popBranchless(Item& item) {   //almost branchless!
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i].topId() == m_readId);

    m_cyclicQueues[idx].pop(item);
    const bool isRealQueue = idx > 0;
    m_readId += isRealQueue;
    return isRealQueue;
}
```

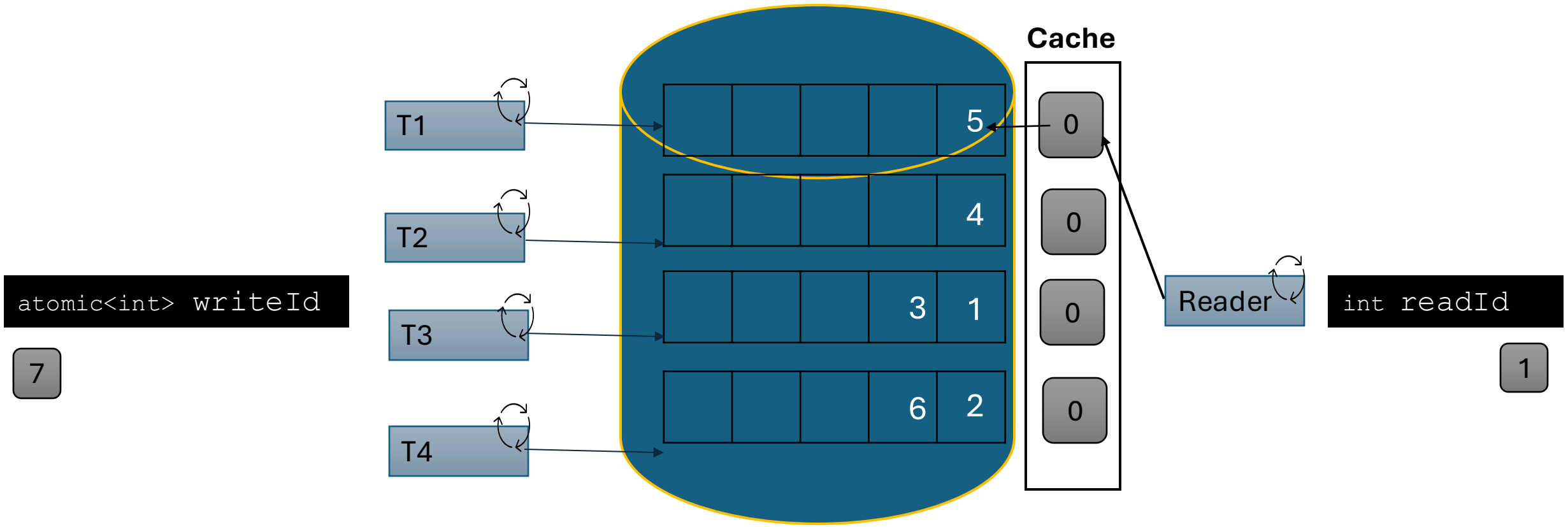
Removing the last branch

Install Dummy queue at `m_cyclicQueues[0]`

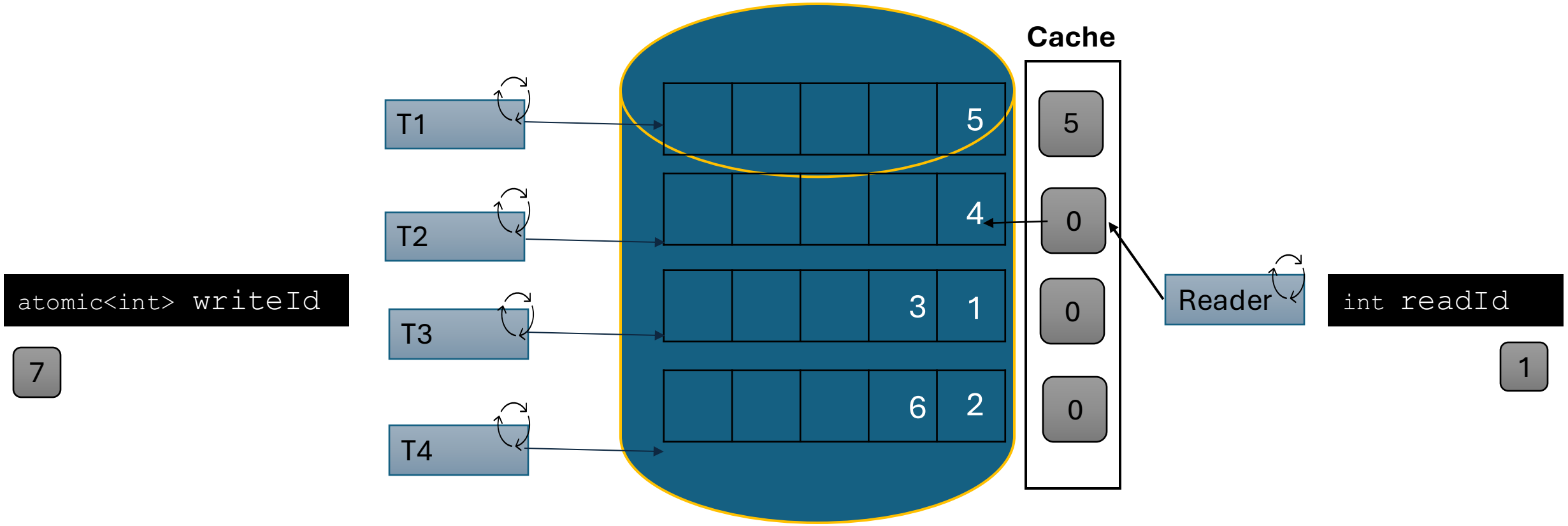
```
bool SCLFQ::popBranchless(Item& item) {
    int_fast32_t idx{0};
    for (uint_fast32_t i = 1; i <= ProducersCount; ++i)
        idx += i * (m_cyclicQueues[i].topId() == m_readId);

    m_cyclicQueues[idx].pop(item);
    const bool isRealQueue = idx > 0;
    m_readId += isRealQueue;
    return isRealQueue;
}
```

Change of behavior!

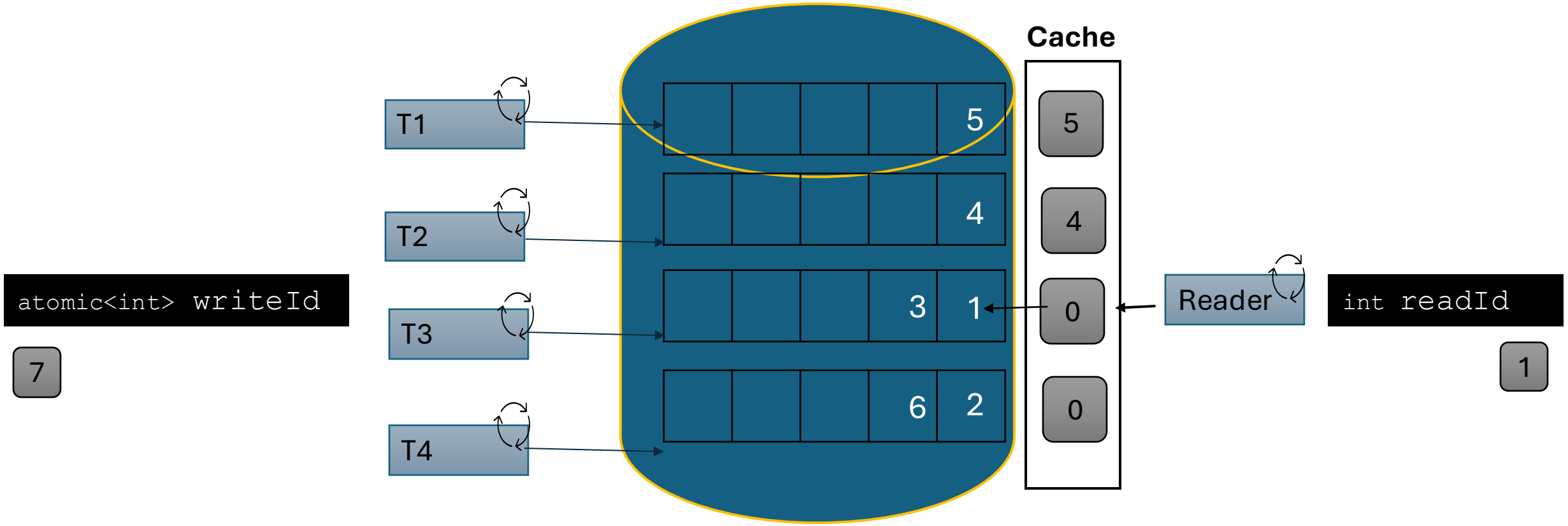
Index cache


Cache
T1
T2
T3
T4
5
4
3
1
6
2
0
0
0
0
Reader
atomic<int> writeId
7
int readId
1

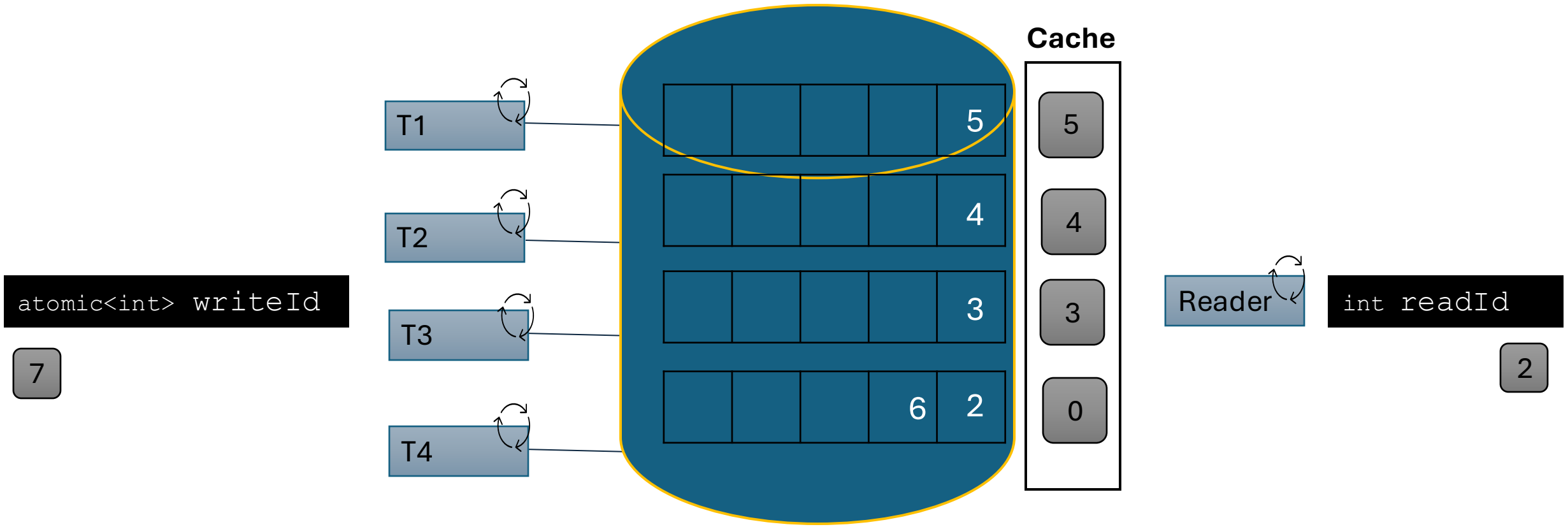
Index cache


Cache
T1
T2
T3
T4
5
4
3
1
6
2
5
0
0
0
Reader
atomic<int> writeId
7
int readId
1

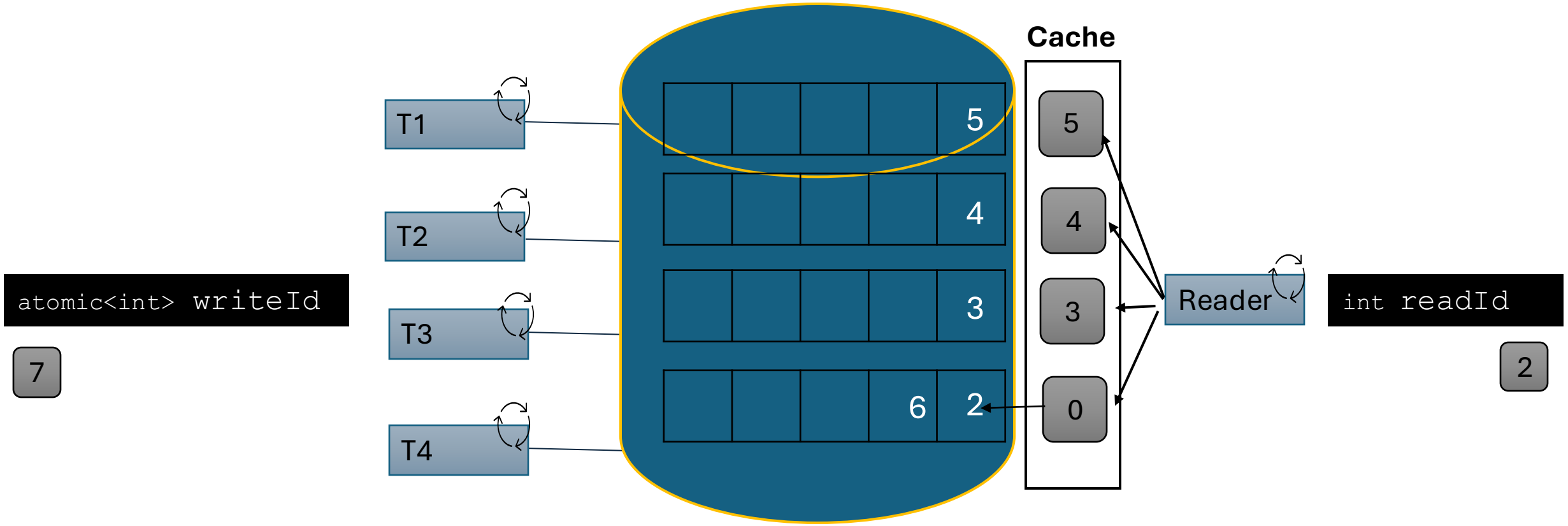
Index cache


Cache
T1
T2
T3
T4
5
4
3
1
6
2
5
4
0
0
Reader
atomic<int> writeId
7
int readId
1

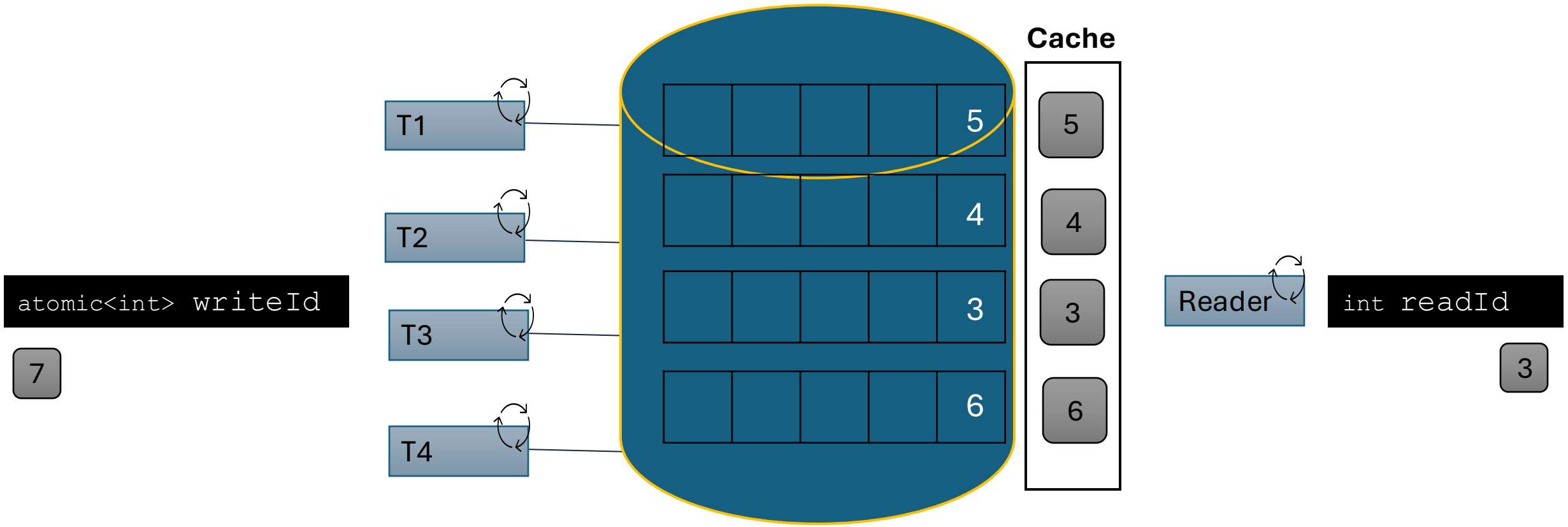
Index cache


Cache
T1
T2
T3
T4
5
4
3
6
2
5
4
3
0
atomic<int> writeId
7
Reader
int readId
2

Index cache


Cache
T1
T2
T3
T4
5
4
3
6
2
5
4
3
0
Reader
atomic<int> writeId
7
int readId
2

Index cache


Cache
T1
T2
T3
T4
5
4
3
6
5
4
3
6
atomic<int> writeId
7
Reader
int readId
3

Benchmarking time!

Micro-benchmark is cool, But…

- Make sure you measure the correct thing
- Make sense of the results
- Always measure your app, and in a real scenario

Benchmarking time!

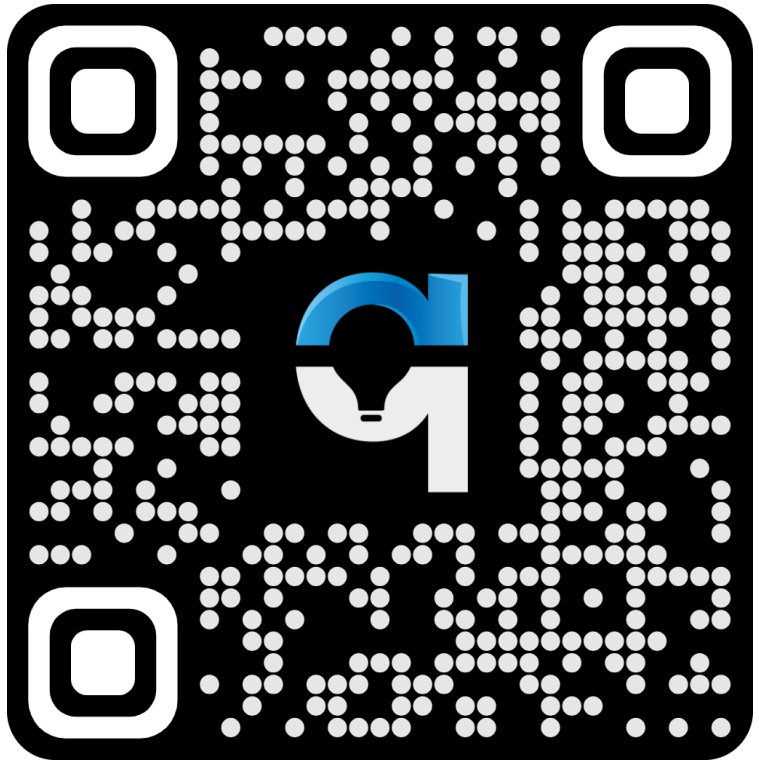
What to measure:

- Time in the queue
- Event latency
- Queue size
- NW buffer utilization

Aggregation:

- Average / median
- 95 / 99 / 99.9 percentile
- Min / Max

Source code QR/link



https://gitlab.com/qspark-public/sclfq

We are hiring!

qspark.co/opportunities