Concurrent Data Structures for Modern Hardware

Fedor G Pikus

Coding Examples

- Benchmarks and concurrent data structures: https://github.com/fpikus/ConcurrentCpp
- Benchmarks use Google Benchmark https://github.com/google/benchmark

Designing Concurrent Data Structures

- Concurrent programs are hard to write in general
  - Even harder if you want correct concurrent programs too
- Good design practices are essential:
  - Modularity, good interfaces, separation of concerns
- Parallel algorithms vs thread-safe data structures:
- Design algorithms to operate on data in parallel
  - Even better, avoid operating on the same data concurrently
- Design data structures to provide access to the data and the necessary concurrency guarantees

Designing Concurrent Data Structures

- Transactional interfaces
- Thread-safety guarantees
- Performance as a design goal
  - Performance is defined with specific metrics
- Targeted specification: design only the features you need
  - Use restrictions can be traded for performance and complexity
  - What needs to be fast? What can be slow?
  - Can access be limited?
  - What are memory requirements?

High-Performance Concurrent Data Structures

- The best synchronization mechanism for each domain:
  - Lock-free or lock-based
- Who does the work:
  - Eager, lazy, helping (threads cooperate)
- Synchronization between domains:
  - DCLP and similar protocols
- How to make new data visible? And old data not visible?
  - Publishing protocol, and “that’s the hard one” (exploit restrictions!)
- How to reclaim unused memory?
  - The hardest part (exploit restrictions! – do nothing?)

Lock-Free vs Lock-Based

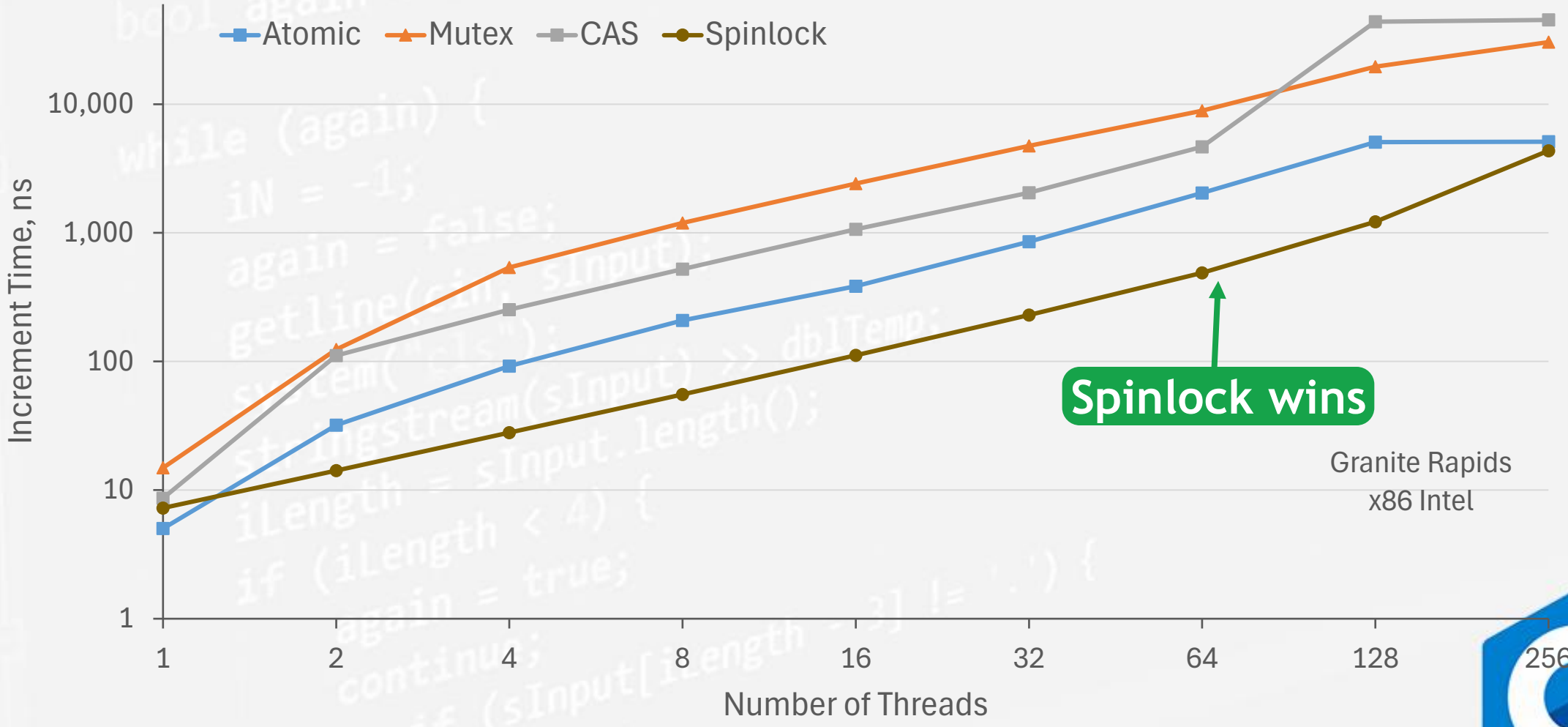
Is Lock-Free Always Better?

- In a wait-free program, all threads are making progress
  - At all times – that's as fast as it can be
- In a lock-free program, at least one thread is making progress
  - Possibly more than one, but usually not all – slower
- In a lock-based program, one thread is making progress
  - Even that is not guaranteed – the slowest
- Locks bring deadlocks, livelocks, starvation, and preemption
  - Lock-free is not as innocent as many people think but the above guarantees hold true (for some value of true)

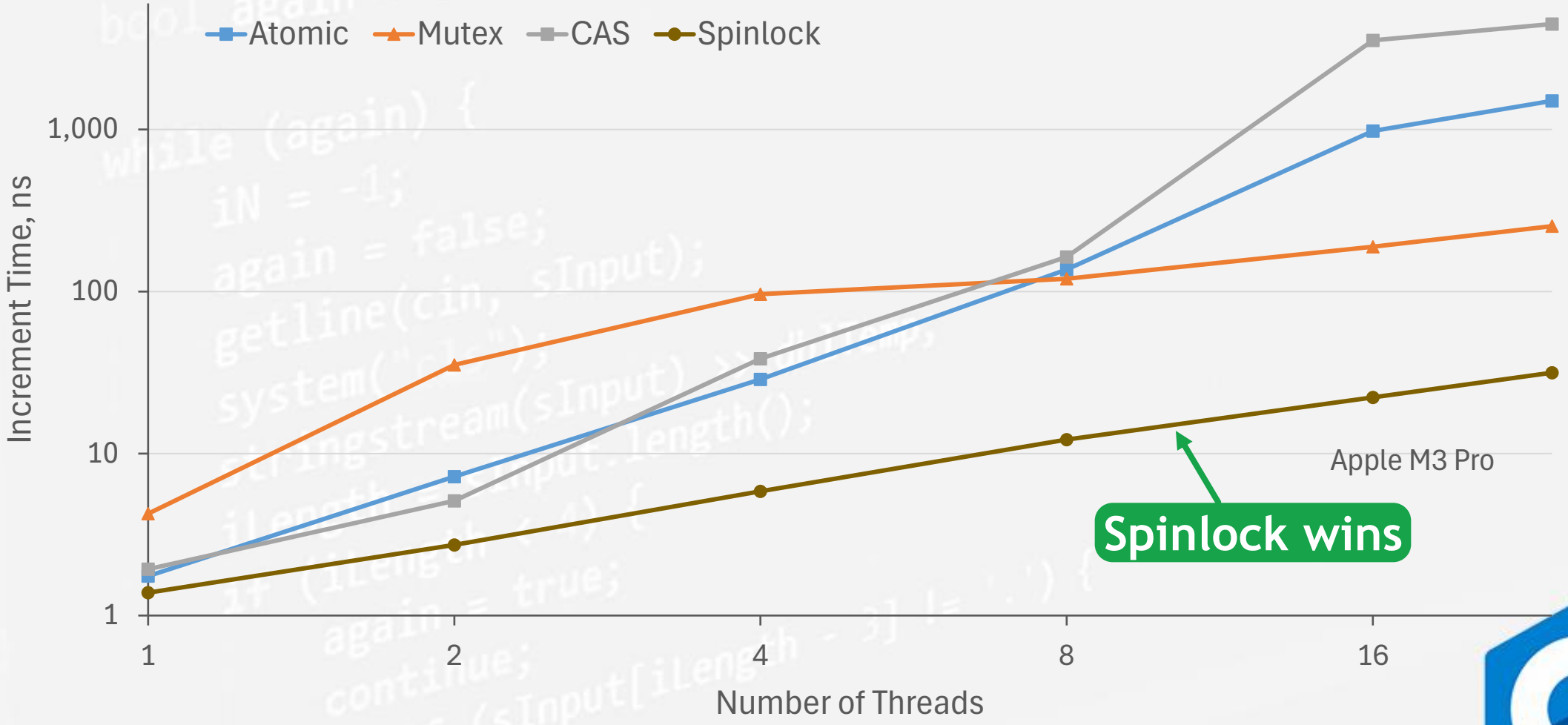
Is Lock-Free Always Faster?

- Benchmark: increment a shared integer on all threads
  - Comparing wait-free vs lock-free is usually artificial
- Maximum contention – the hardest stress test
  - Whoever wins this can handle anything

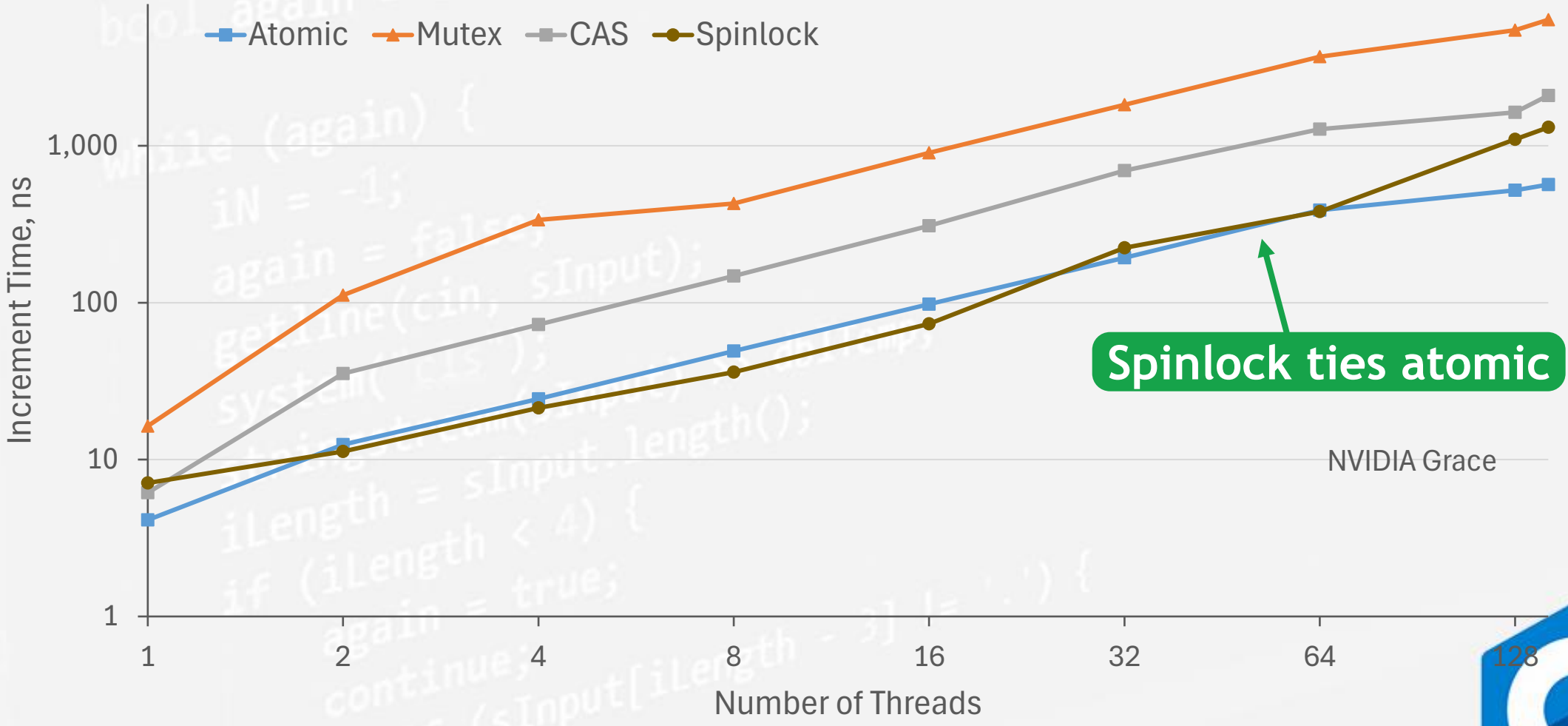
Is It?


Atomic
Mutex
CAS
Spinlock
Increment Time, ns
10,000
1,000
100
10
1
1
2
4
8
16
32
64
128
256
Number of Threads
Spinlock wins
Granite Rapids
x86 Intel

Is It Hardware-Dependent?

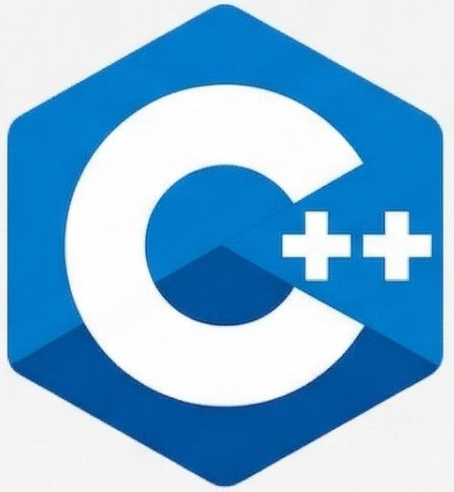


Is It ARM vs x86?



Is Lock-Free Faster?

- Under high contention, spinlock usually outperforms atomics
- Lock-free (CAS) is slower than wait-free
  - But very few computations can be wait-free
- Some locks are faster than others
- What makes a good spinlock?



What Is the Best Lock?

How to Write Good Spinlock

```
class Spinlock {
  std::atomic<int> flag_ {};  // 1 - locked
  void lock() {
    while (flag_.load() || flag_.exchange(1))
      backoff();
  }
  void unlock() { flag_.store(0); }
};
```

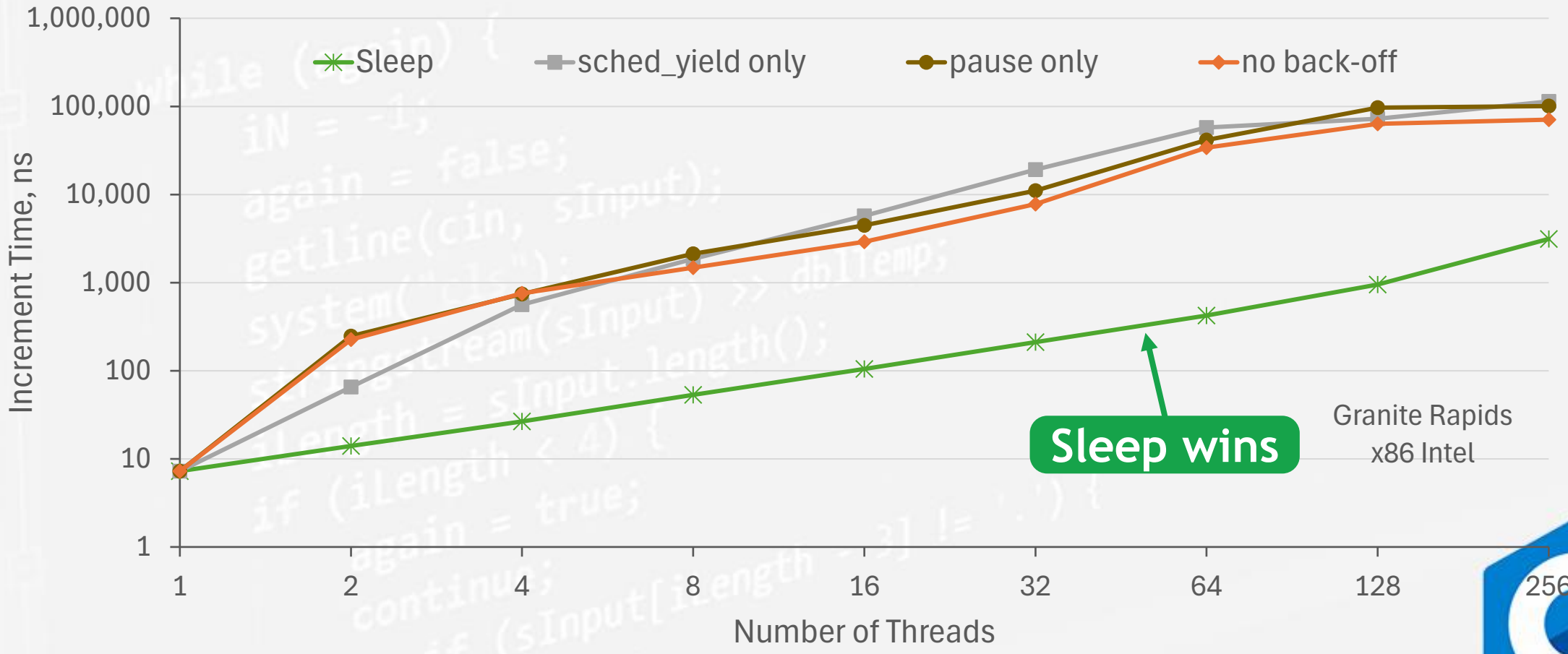
No cache lock

Lock

Yield scheduler

Unlock

Back-Off at High Contention



Why Locks Are Faster

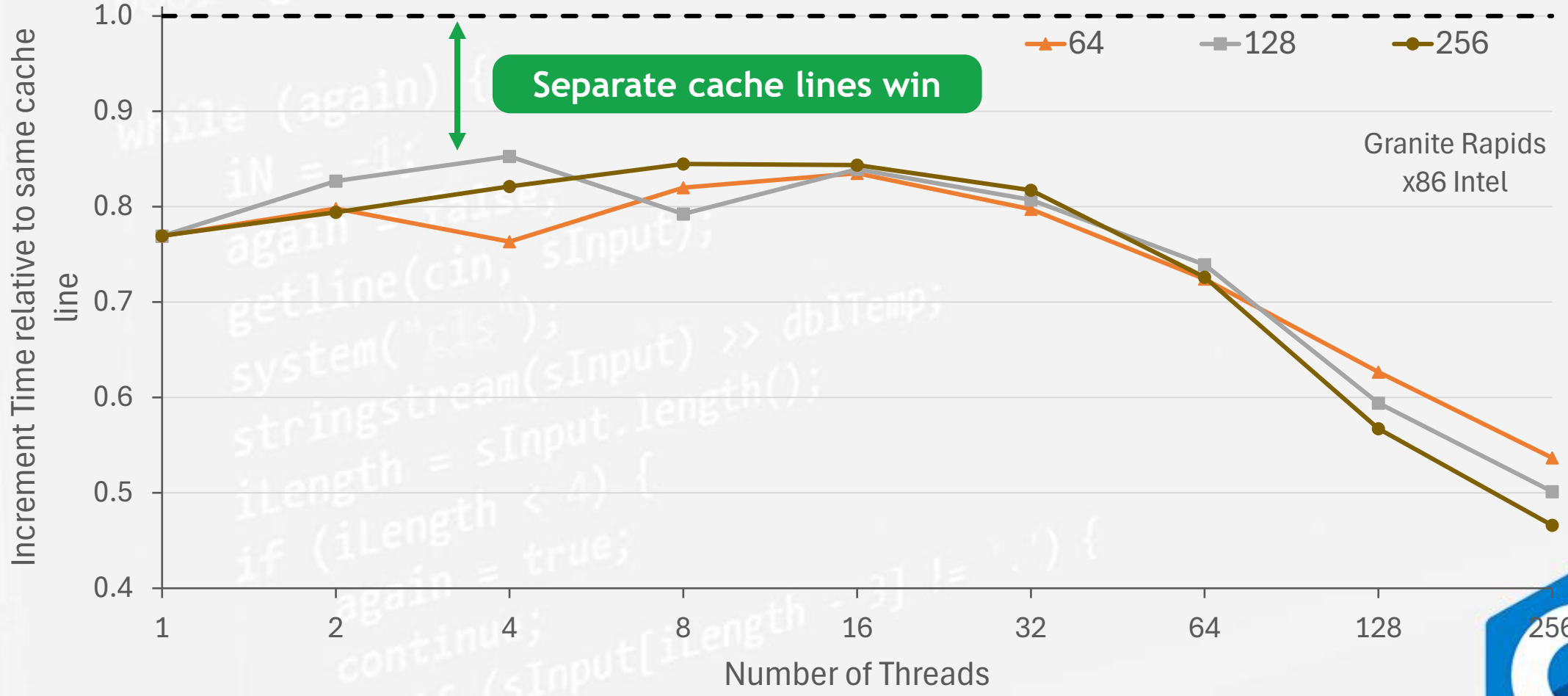
- At high contention, cache coherency (MESI) overhead is critical
- When lock owner wants to write 0 to release the lock…
  - Cache is in M or E state: just write it
  - Cache is in S state:
    - Send Invalidate broadcast to all cores and wait for acknowledgement
    - Upgrade from S to E state and write (state changes to M)
- Rapid backoff is essential for lock performance
  - Yield scheduler after few failures to lock
  - Experimentally `nanosleep(1ns)` gives best performance
- Aggressive backoff: fewer active reads – fewer invalidations

Optimizing Critical Section

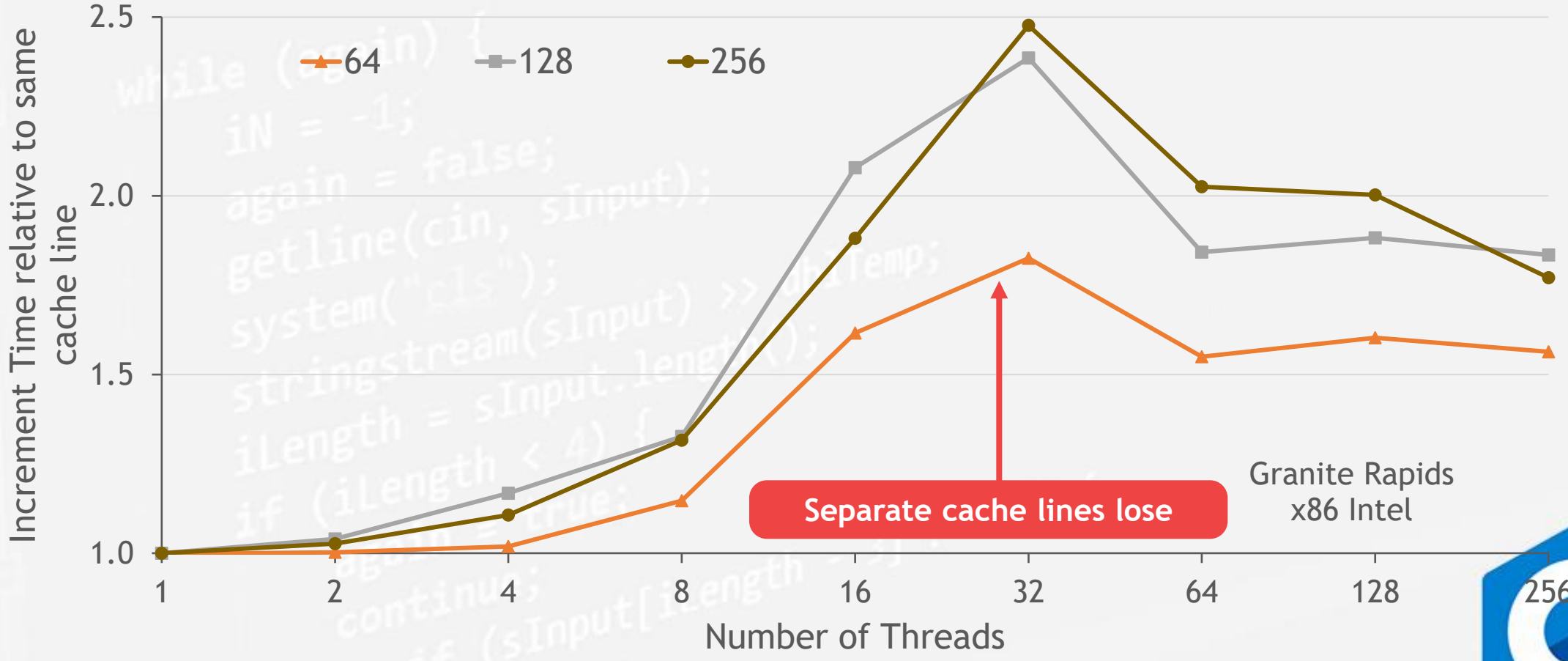
```
unsigned long x;
spinlock s;                 // Guards x
…
lock_guard g(s);
++x;
```

- Do spinlock and shared data go on the same cache line?
- Duh – of course, the core that has the lock already fetched that cache line, anything else is a cache miss

Measure, Don't Guess

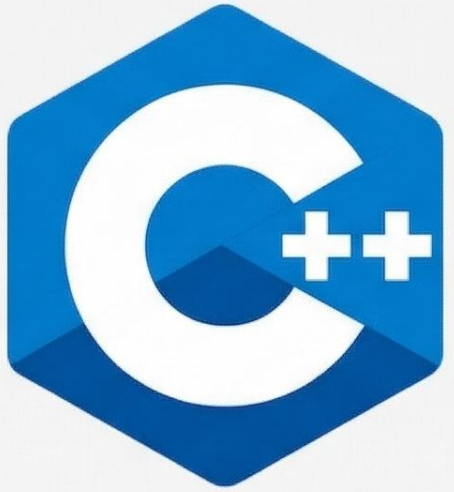


Critical Section – Low Contention



Optimizing Critical Section

- At high contention, spinlock and payload should be separate
  - No core owns spinlock cache line for long
  - Other cores read the lock – cache line becomes Shared (S)
  - Payload on separate cache line – no competition for that line
  - True only for writer workloads
- At low contention, spinlock and payload share cache line
  - Lock owner (usually) owns the cache line
  - Payload on the same cache line avoids a cache miss



Are Locks Really Faster?

Are Locks Really Faster?

- We just measured it!
- At the highest contention
- The most challenging case
- If locks are better here, they are better everywhere
- The first rule of performance…

Are Locks Always Faster?

- Real programs access shared data “in service” of the real work
  - High-contention sharing happens but not 100% of the time
  - Best case for shared increment is serial execution (zero overhead)
- We can model it by mixing operations on local and shared data

```
unsigned long x;
spinlock s;
for (…) {
  unsigned long xl = 0;
  xl = SharedUpdate(x, xl);
  xl = LocalWork(xl);
}
```

Shared

Per thread

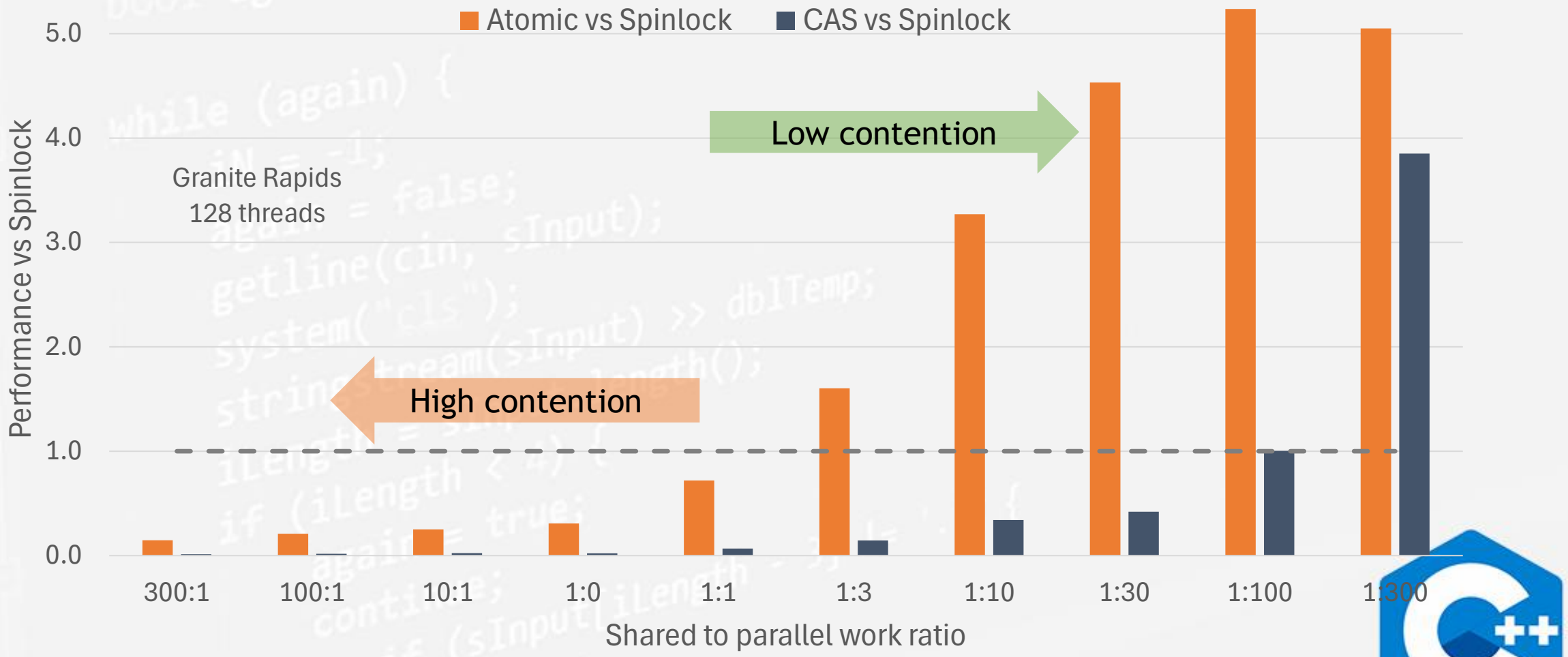
Benchmarking Synchorization Overhead

```
unsigned long n = 0;
for (…) {
  unsigned long dx = 1;
  for (size_t i = 0; i != nshared; ++i) {
    x.fetch_add(dx, std::memory_order_relaxed));
  }
  double y = x;
  for (size_t i = 0; i != nwork; ++i) {
    y = sin(++n + cos(y));
  }
  dx = y + 1;
}
```

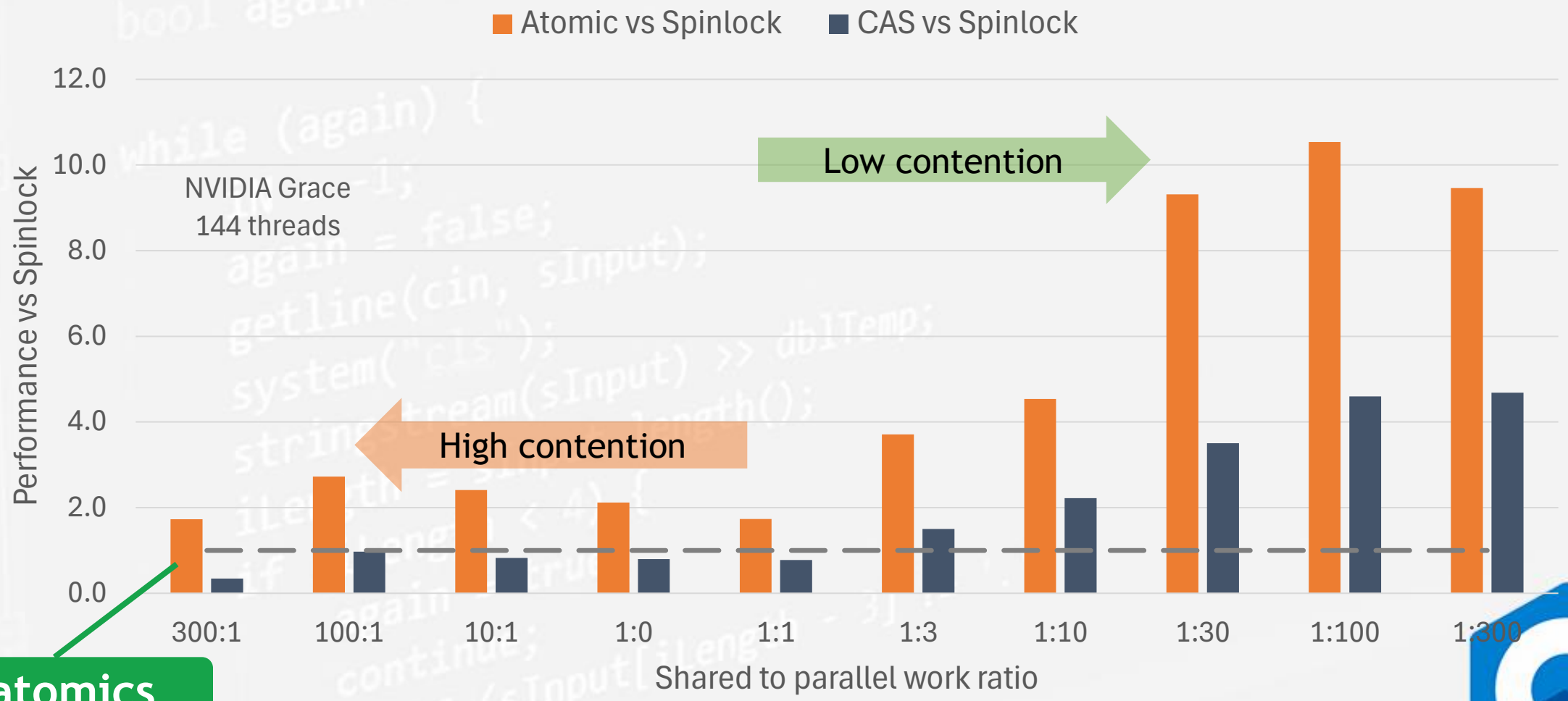
Per thread

- n is the measure of work on this thread
  - Depends on the ratio of shared and local computing
- Use lock or CAS loop instead of `fetch_add` to benchmark other synchronization options

Good, Better, Best?

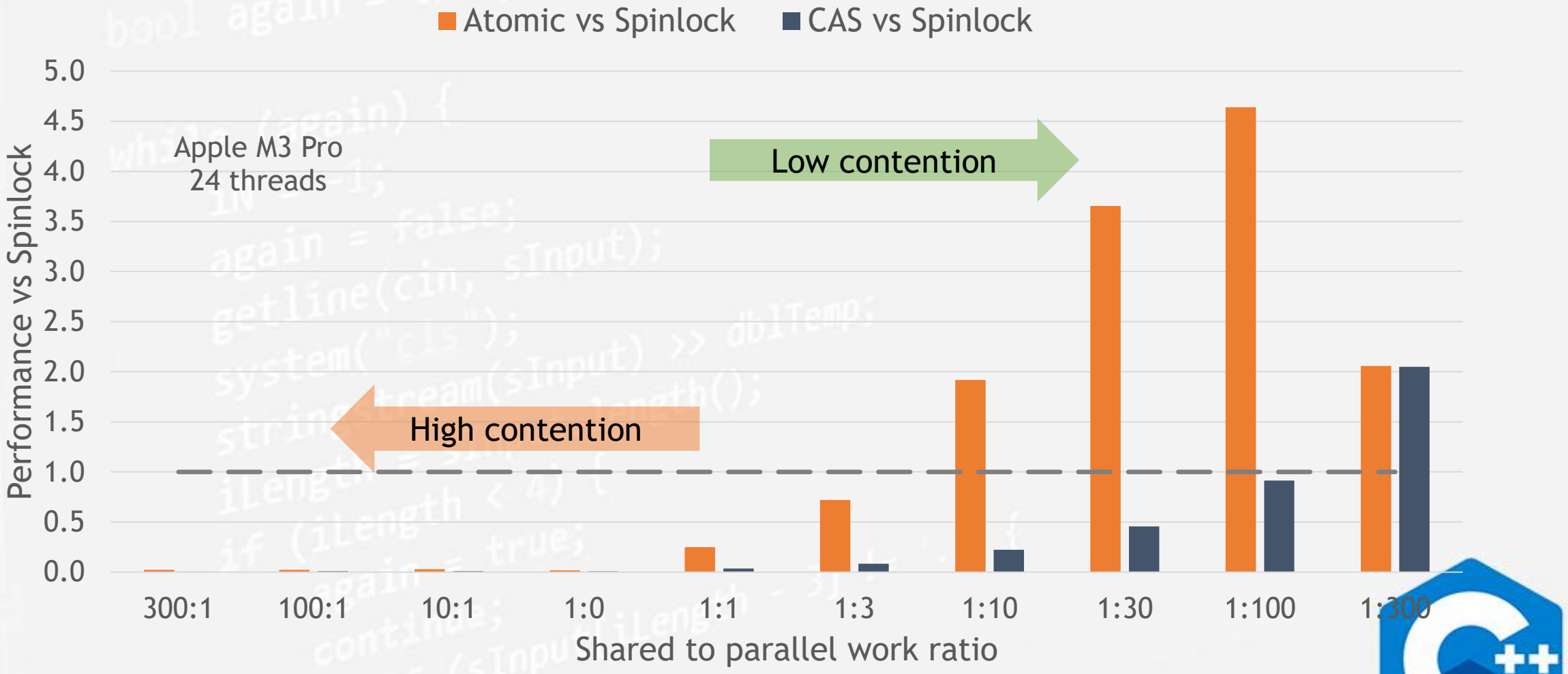


ARM Has Out-of-Order Stores but…



far atomics

ARM Has Out-of-Order Stores but…



Why?!

- At high contention, spinlock is the best
  - Better than the single atomic instruction (increment)
  - Aggressive backoff is the main reason (reduces invalidation traffic)
- At low contention, spinlock is the slowest
  - Throughput degradation out of proportion to the shared portion
  - 10:1 parallel/shared ratio → 3x performance degradation vs atomic
  - Spinlocks aren't just slow – they degrade performance of surrounding code which has no shared data
- How? Why?

Perils of Implicit Synchronization

- Locks maintain exclusion by establishing happens-before and happens-after relations between independent variables
  - Lock flag and protected data
  - Hardware has oddities when accessing the same locked cache line repeatedly (lock release again!)
- CAS and atomics operate on a single shared variable
  - Fuses control and data into a single instruction
- Implicit Execution Dependency is bad for concurrency

Spinlock “Poisoning”

- Resource starvation (general):
  - ROB capacity is limited and long-latency stores stall it
  - Long-latency instructions occupy limited Load-Store Unit (LSU) slots
- Specific mechanisms:
  - Order-dependent (x86-specific): spinlock stores stall SB
  - Memory clears: spinlock read speculatively executed then invalidated by release-store
  - Test-and-Test-and-Set (TTAS) spinlock is necessary to avoid RMW storm, but it creates a speculation hazard on the read side
- Not enough to explain all measurements in all regimes
  - Results are clear but the mechanism is not…

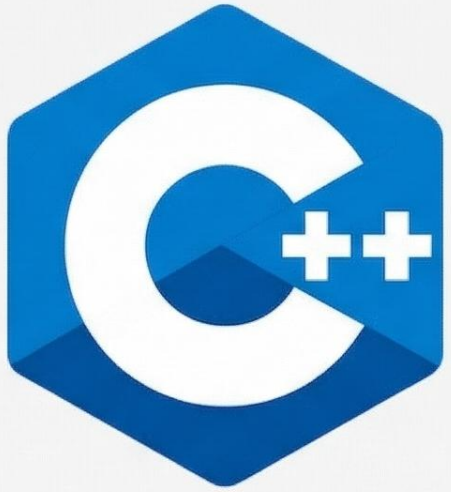
Key Takeaway about Locks

1. **At high contention, a good spinlock beats CAS**
   Usually beats atomics — though fast atomics exist on some hardware

2. **At moderate contention, CAS pulls ahead**
   A single atomic RMW is even faster, until results converge at very low contention

3. **The crossover holds across x86 and ARM**
   Intel, AMD, Neoverse V2, M3 — the point moves, but a crossover always exists

4. **The result is universal — the explanation is not**
   No single mechanism explains this behavior on all hardware



Using Locks Efficiently

Not Everything Is Lock's Fault

- Example – concurrent maximum

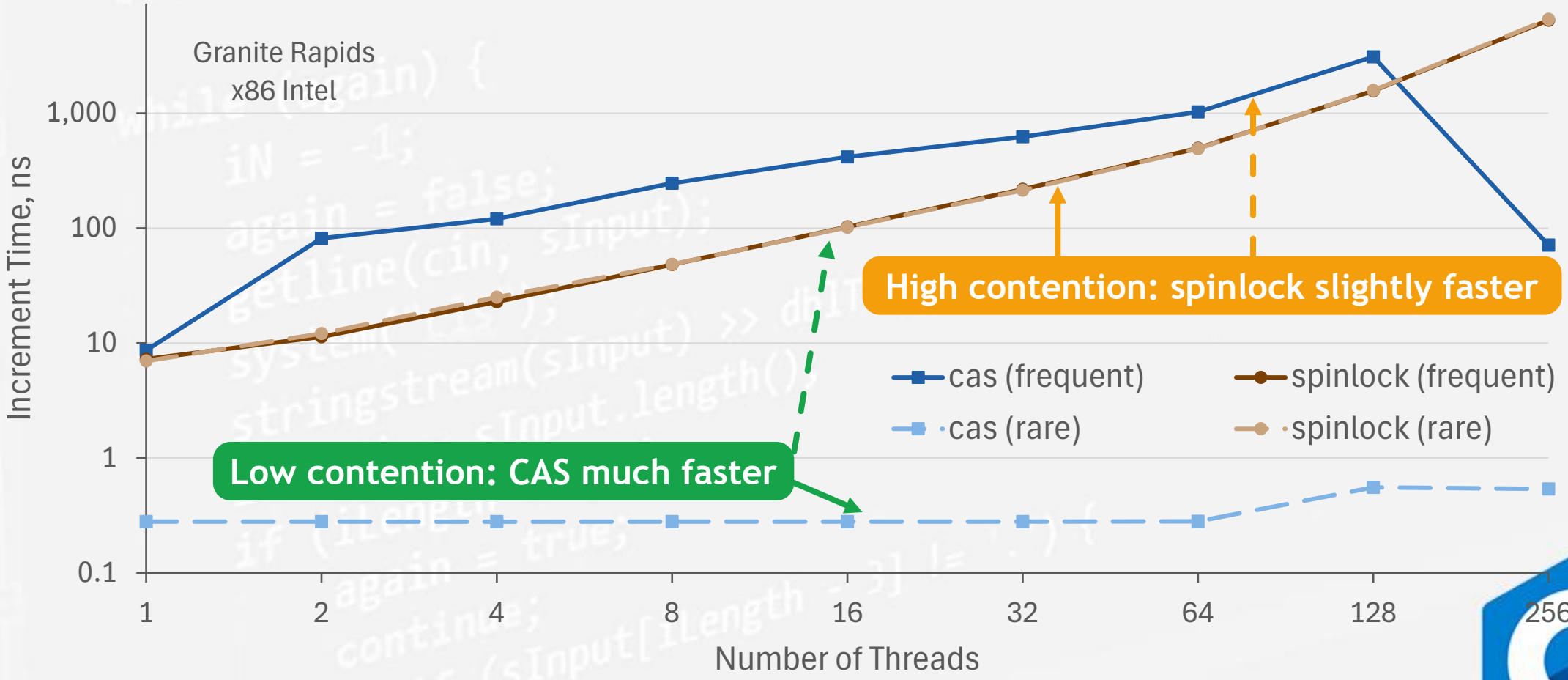
Lock-free (not wait-free)

```
std::atomic<long> nmax;

long n = … new value …
long cur = nmax.load();
while (
   n > cur &&
   !nmax.compare_exchange(cur,n)
){}
```

Lock-based

```
long nmax {};
Spinlock S;
long n = … new value …;

{
   std::lock_guard L(S);
   if (n > nmax) nmax = n;
}
```

Is Lock-Free Faster?



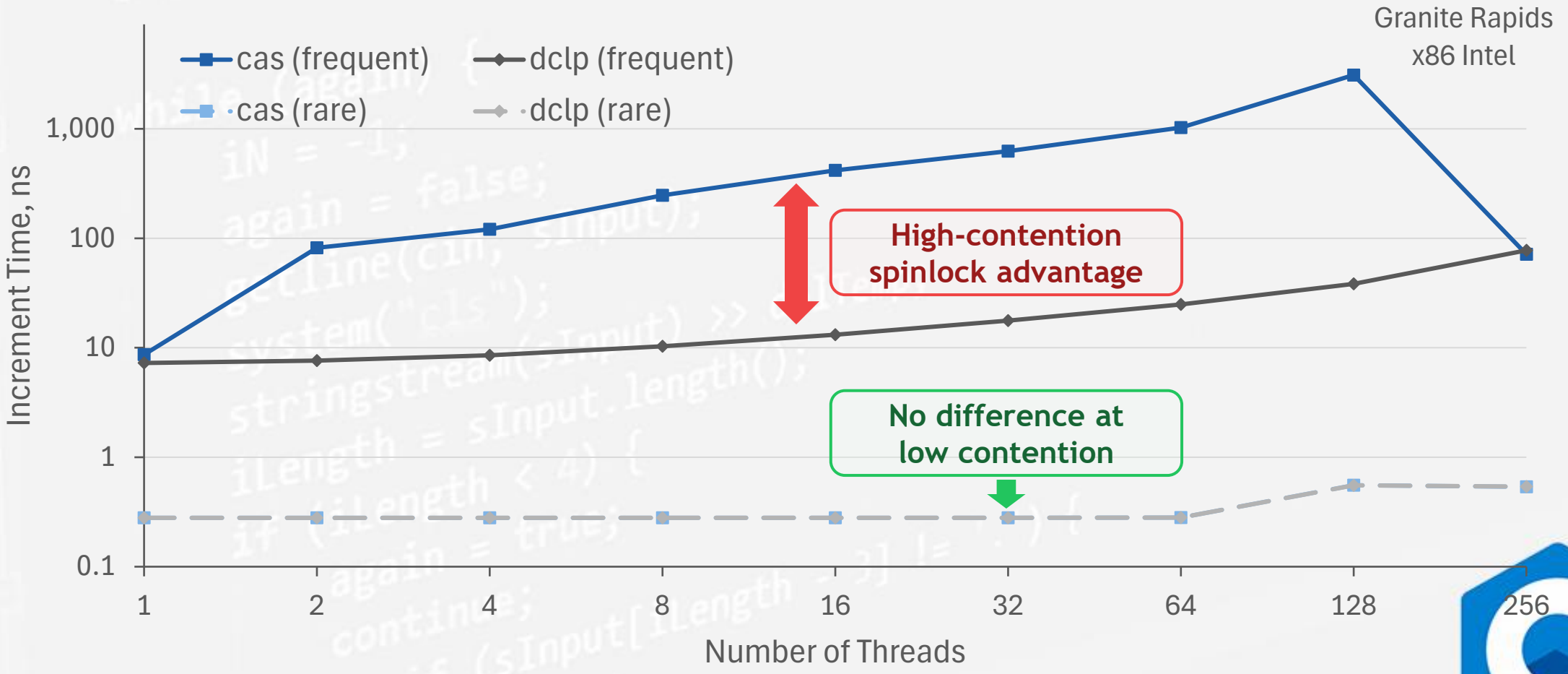
Inefficient Code Is Inefficient

- The problem isn't the locks…

```
std::atomic<unsigned long> nmax {};
Spinlock S;              // Guards (atomic!) nmax
// … on all threads …
unsigned long n = … new value …
if (n > nmax.load(std::memory_order_relaxed)) {
  std::lock_guard L(S);
  if (n > nmax.load(std::memory_order_relaxed))
    nmax.store(n, std::memory_order_relaxed);
}
```

- Double-Checked Locking Pattern (DCLP)

Well-Written Locks Are Fast

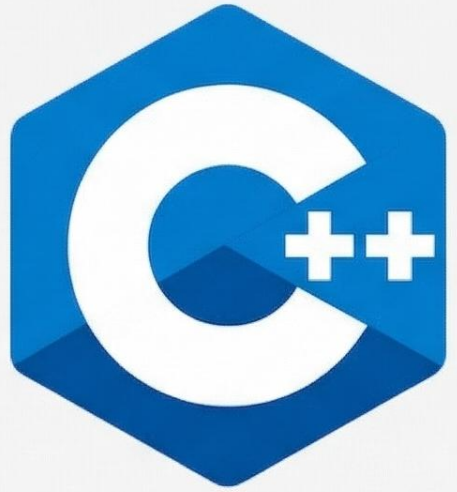


Lock-Free Style, with Locks

- The problem isn't always the locks
- Traditional lock-based code locks too much
  - Concurrent queue: pop head under lock
- You can write CAS-style code with locks
  - Concurrent queue: advance head under lock, copy data in parallel
- Often this results in DCLP (or similar) code:
  - Use atomics and read-only operations outside of lock
  - Critical section protects modification of the atomic variable
    - And nothing else

Main Takeaway

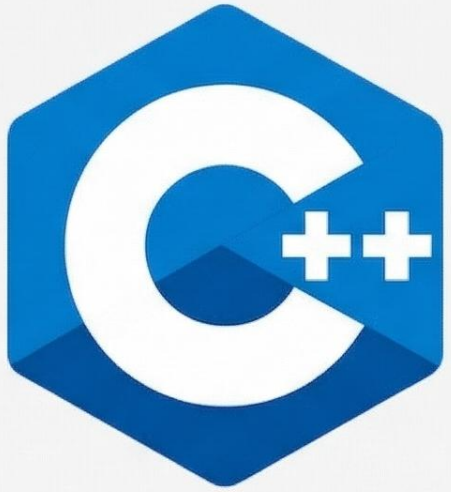
- Locks work better at high contention
  - backoff is essential for high-contention writes
  - Read with retries is critical – need to wait for ACKs on RFO
  - Hidden cost of the lock is its release
- DCLP (“lock-free style”) techniques minimize critical section
- Lock-free works better at low contention
  - Cost of the lock doesn’t matter – disruption of execution flow matters
  - Implicit synchronization: lock variable and guarded variable
  - Complex order restrictions disrupt CPU pipeline
- Locks rule average latency – lock-free rules tail latency



Concurrent Data Structures for Modern Hardware

Basic Principles

- Locks for high-contention domains
  - Unless tail latency or preemption are performance metrics!
  - Remember to optimize critical sections and avoid false sharing
- Lock-free (CAS) for low-contention domains
- Also locks for operations that are hard to synchronize
- Atomic variables for crossing domains
- Publishing protocol for releasing new data
- The hard part is withdrawing and reusing old data
  - What if we don’t?

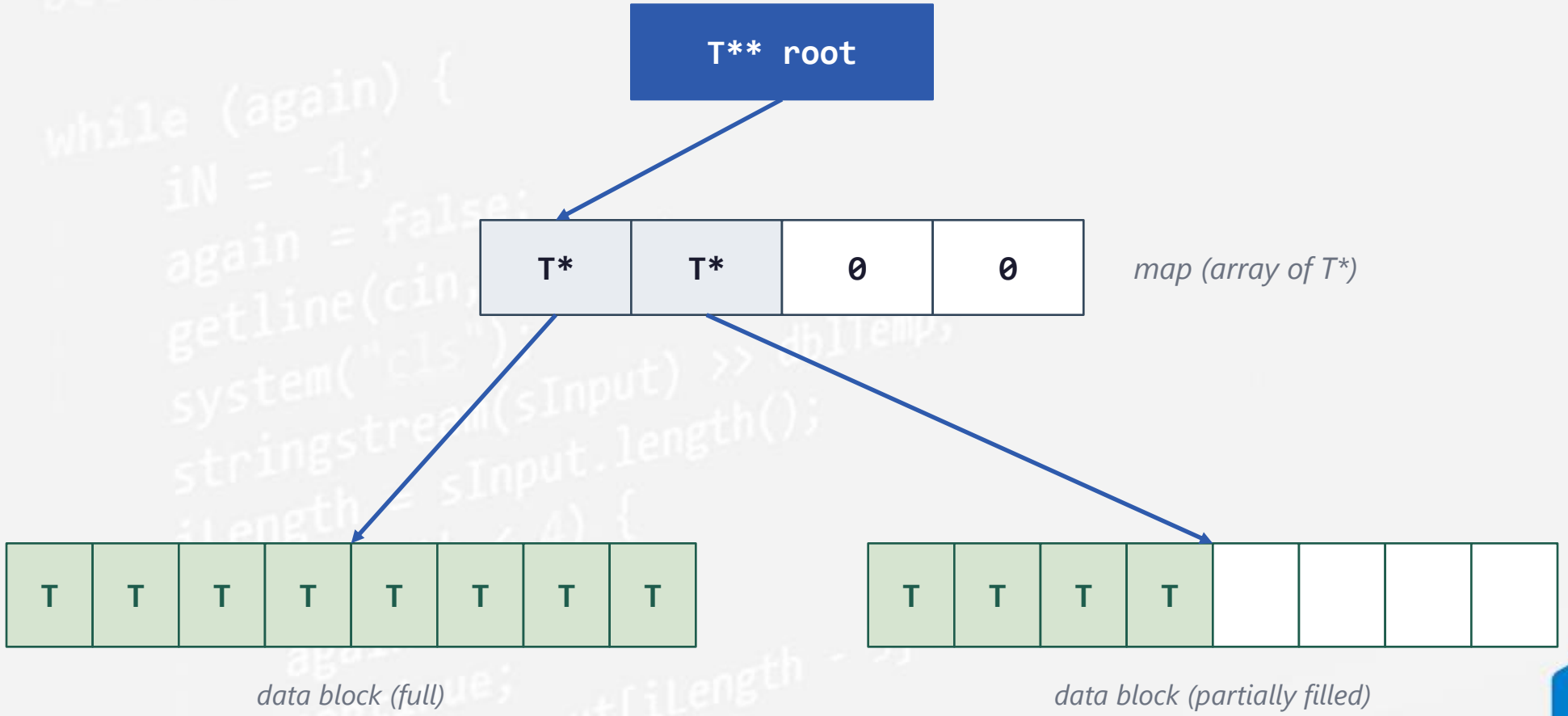


Basics – Data Container

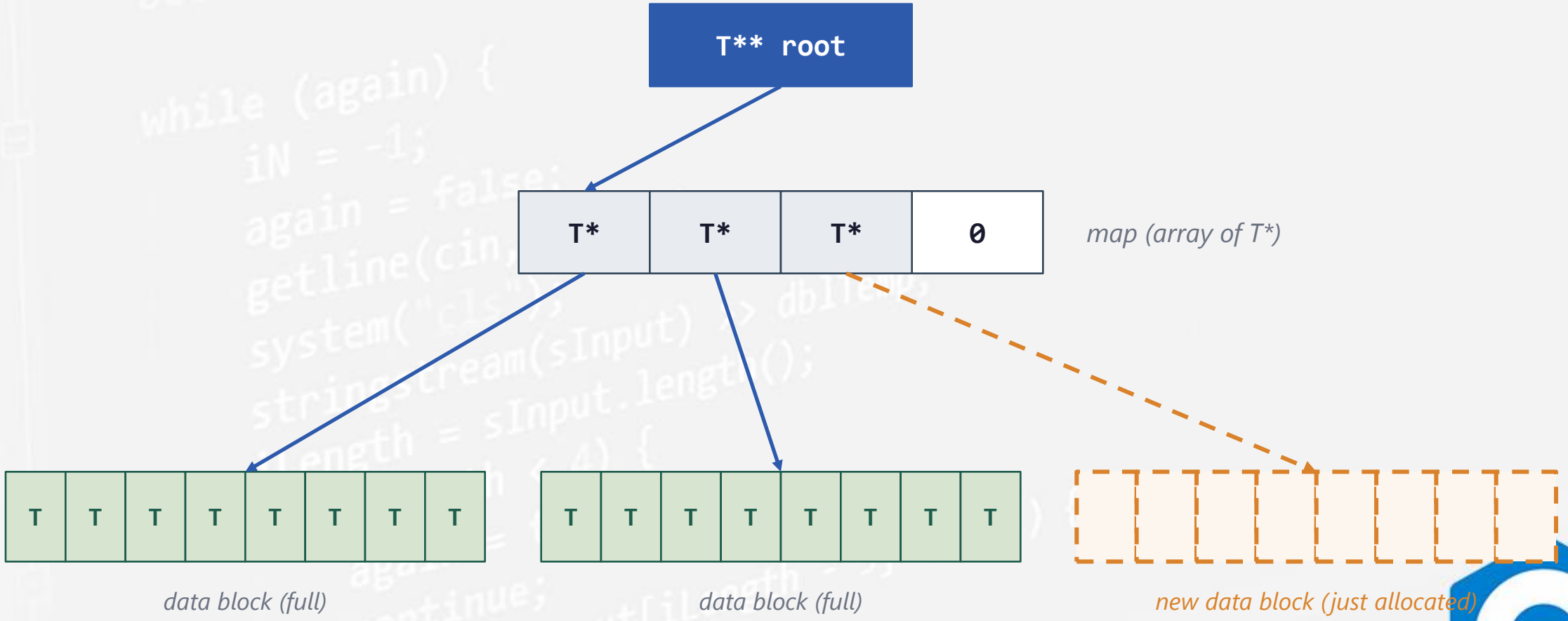
Thread-Safe Growable “Array”

- Frequently needed: thread-safe vector-like container
  - Increasing size is thread-safe
  - Accessing elements is thread-safe
  - Concurrent access to the same element is client’s problem
  - No resizing down! Can’t be certain that deletion is safe
- Not std::vector – reallocation moves data memory
- Deque? Growth does not invalidate existing elements
- Not std::deque – reallocation moves internal memory
  - Ideally, all calls to `operator[]` run concurrently and non-blocking
- Custom deque, then

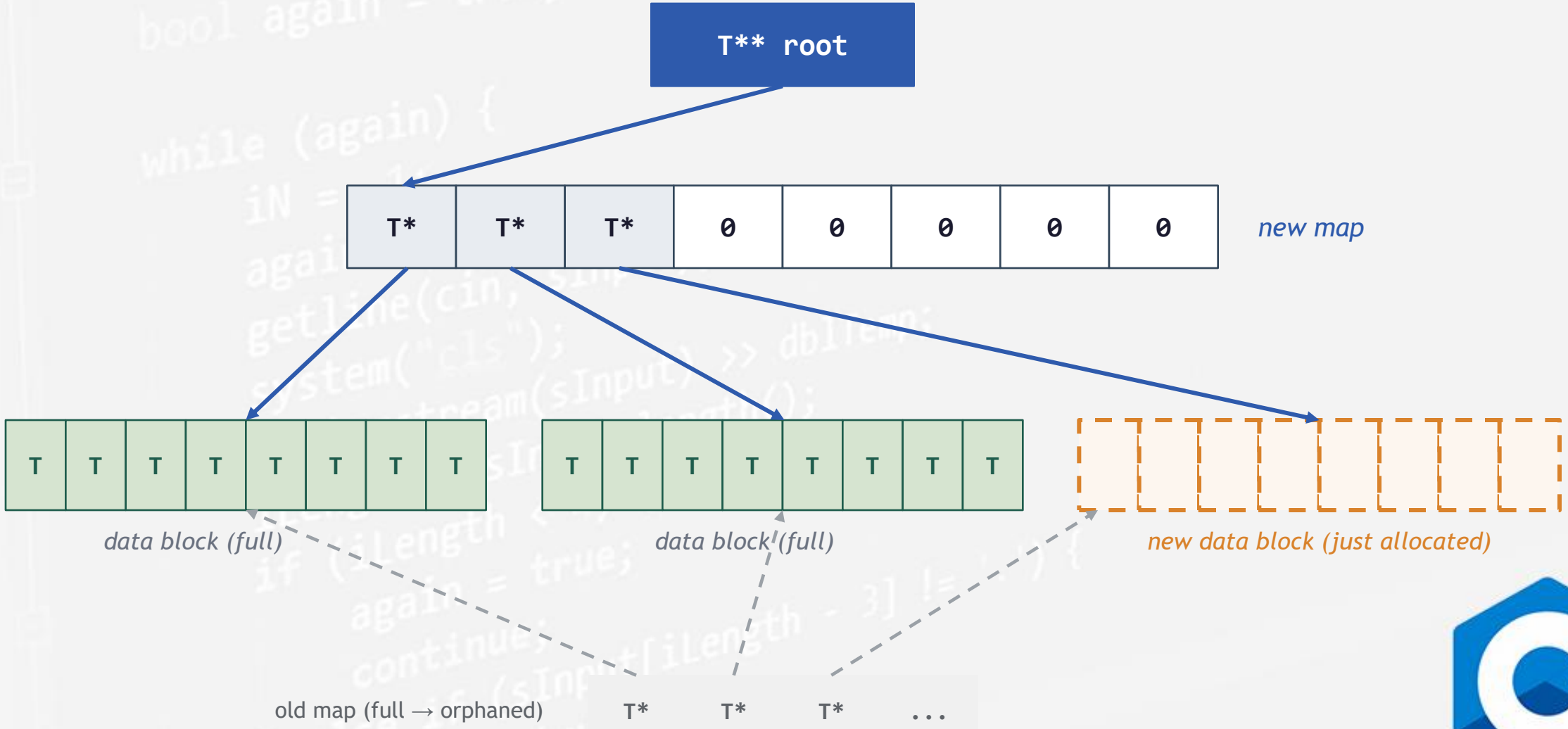
Deque Structure



Growing a Deque



Why Not std::deque?



Concurrent Deque

- Data is block-allocated and never moves
- Atomic Root pointer: all accesses begin from the root
  - Root can change (atomically) to point to new map
- Nothing is ever deleted:
  - Resize to shrink not supported (other threads may access “old” data)
  - Old maps cannot be safely collected until destruction (minor issue)
- Map pointers are not atomic and never change once created

Concurrent Deque Use Cases

- Each thread reserves some number N of elements
  - Resize up is thread-safe
  - Each thread uses its range of elements, or
  - Client arranges exclusive (write) access externally
- Any thread can check current size
  - This thread can access all elements in range
  - Client arranges exclusive (write) access externally
- Element access is low contention and wait-free (atomics)
- Resize is rare but hard to do without locks, so spinlock

Concurrent Deque - Interface

```
template <typename T> class Deque {
  public:
  Deque(…) {…}
  void resize(size_t new_size);
  T& operator[](size_t i);
  const T& operator[](size_t i) const;
  size_t size() const;
};
```

Standard stuff

Locking: [0, new_size) visible

Non-locking

Non-locking

Non-locking: [0, size()) visible

Concurrent Deque - Data

```
template <typename T> class Deque {
  alignas(64) mutable spinlock s_;
  alignas(64) std::atomic<size_t> size_ {};
  alignas(64) std::atomic<T**> map_ {};
};
```

Some const methods lock

Atomic handshake

Root pointer

Concurrent Deque - Fast Path

- Call `size()` and use elements known to exist - wait-free!

```
template <typename T>
size_t Deque<T>::size() const {
  return size_.load(std::memory_order_acquire);
}
template <typename T>
T& Deque<T>::operator[](size_t index) {
  T** m = map_.load(std::memory_order_acquire);
  size_t block_idx = index >> BlockShift;
  size_t local_offset = index & BlockMask;
  return m[block_idx][local_offset];
}
```

Concurrent Deque - Resize

```
template <typename T>
void Deque<T>::resize(size_t new_size) {
  size_t cur_size = size_.load(acquire);
  if (new_size <= cur_size) return;
  std::lock_guard lock(s_);
  cur_size = size_.load(relaxed);
  if (new_size > cur_size) { … grow … }
}
```

Slideware

DCLP!

Concurrent Deque - Resize

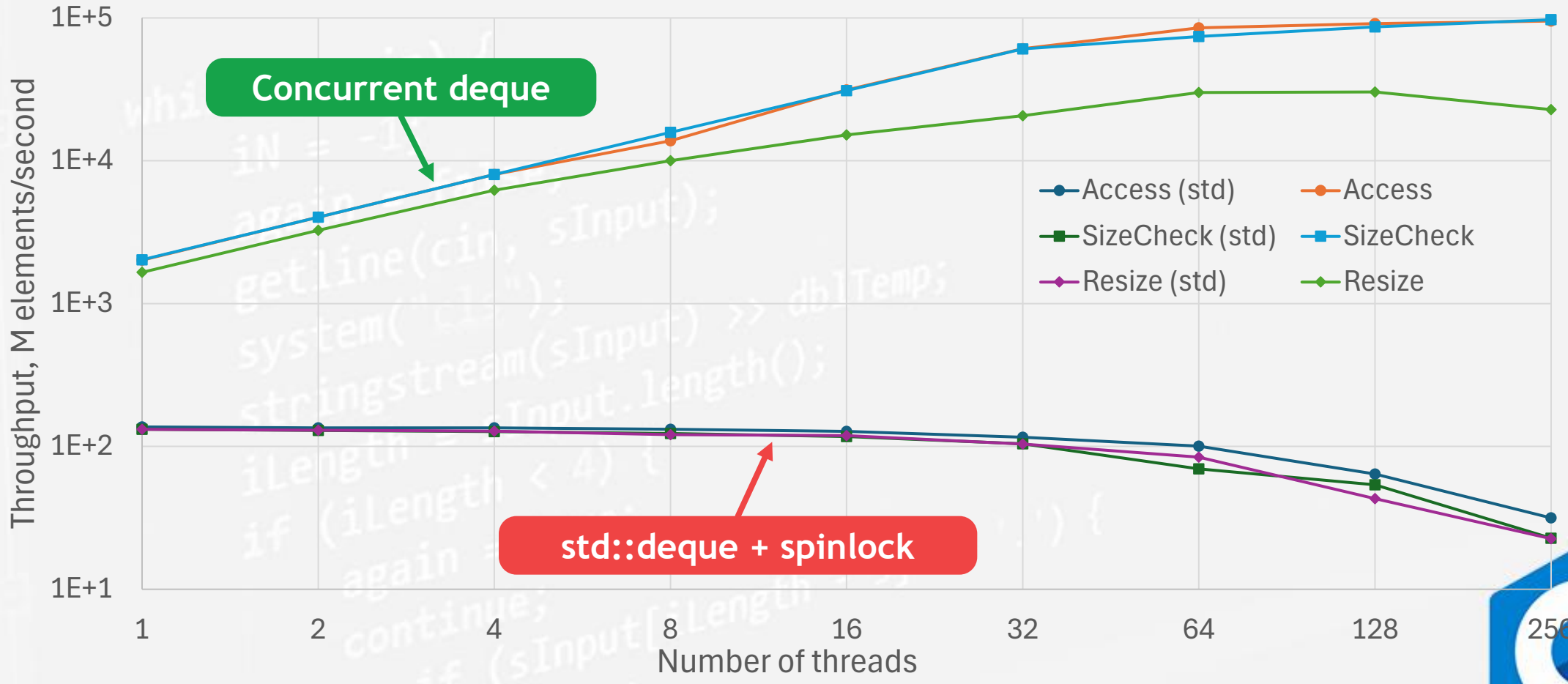
```
std::lock_guard lock(s_);
if (new_size > cur_size) {
  if (… map is full …) {
    … create new_map, old map→garbage list …
    map_.store(new_map, release);
  }
  … create new blocks …
  size_.store(new_size, release);
}
```

Under lock from now on

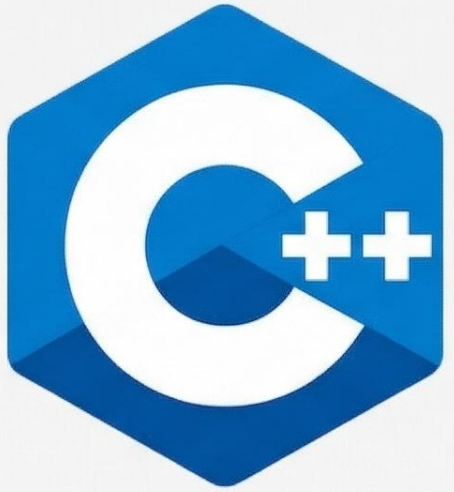
Publish

unlock is release!

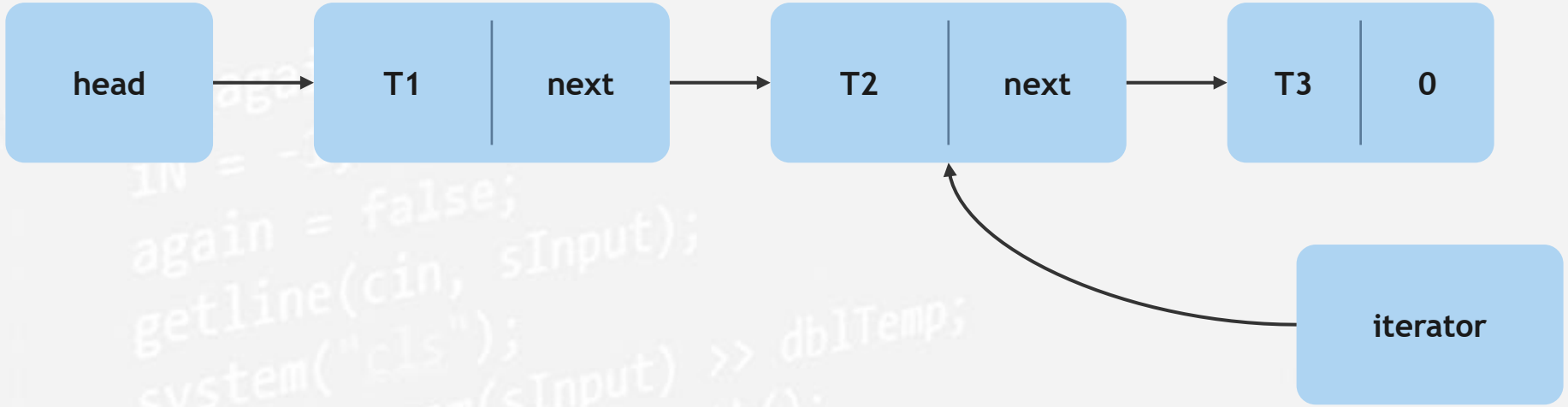
Benchmarks vs Spinlocked std::deque



Lock-Free List



Forward (Single-Linked) List



- Interface: insert or erase after a given node or at the head
- Iterate over the list (for example, find an element)
- STL has it as `std::forward_list`

Thread-Safe Forward List

- Wrap `std::forward_list` in a lock?
  - Insert and erase are safe
  - Only one at a time can execute even when they don't interact
  - `begin()/end()` are "safe" but what locks iterator access?
  - `std::find` is not safe – not locked while iterators advance
  - The only way is to hold a lock for the entire duration of `find`
  - With read-write locks, multiple `find` calls can be concurrent but…
  - Each `find` locks out all mutations until it's done – O(N)!

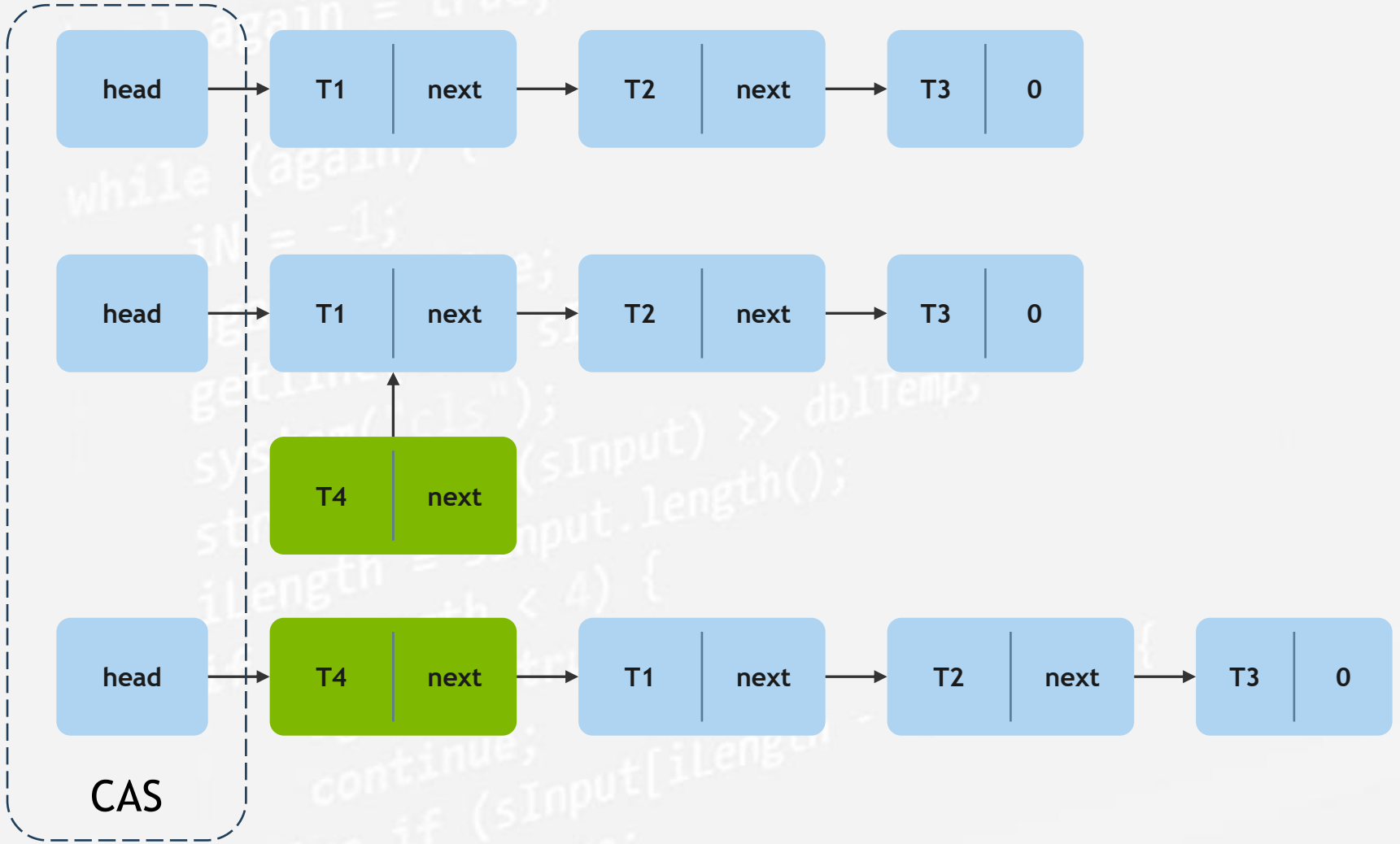
Thread-Safe Forward List

- Lock each node when it's modified?
  - Erase modifies two nodes – erased one and the one pointing to it
  - Advancing an iterator needs to lock two nodes, one to read the next pointer and one to make sure the iterator does not dangle
  - Probably will deadlock somewhere…

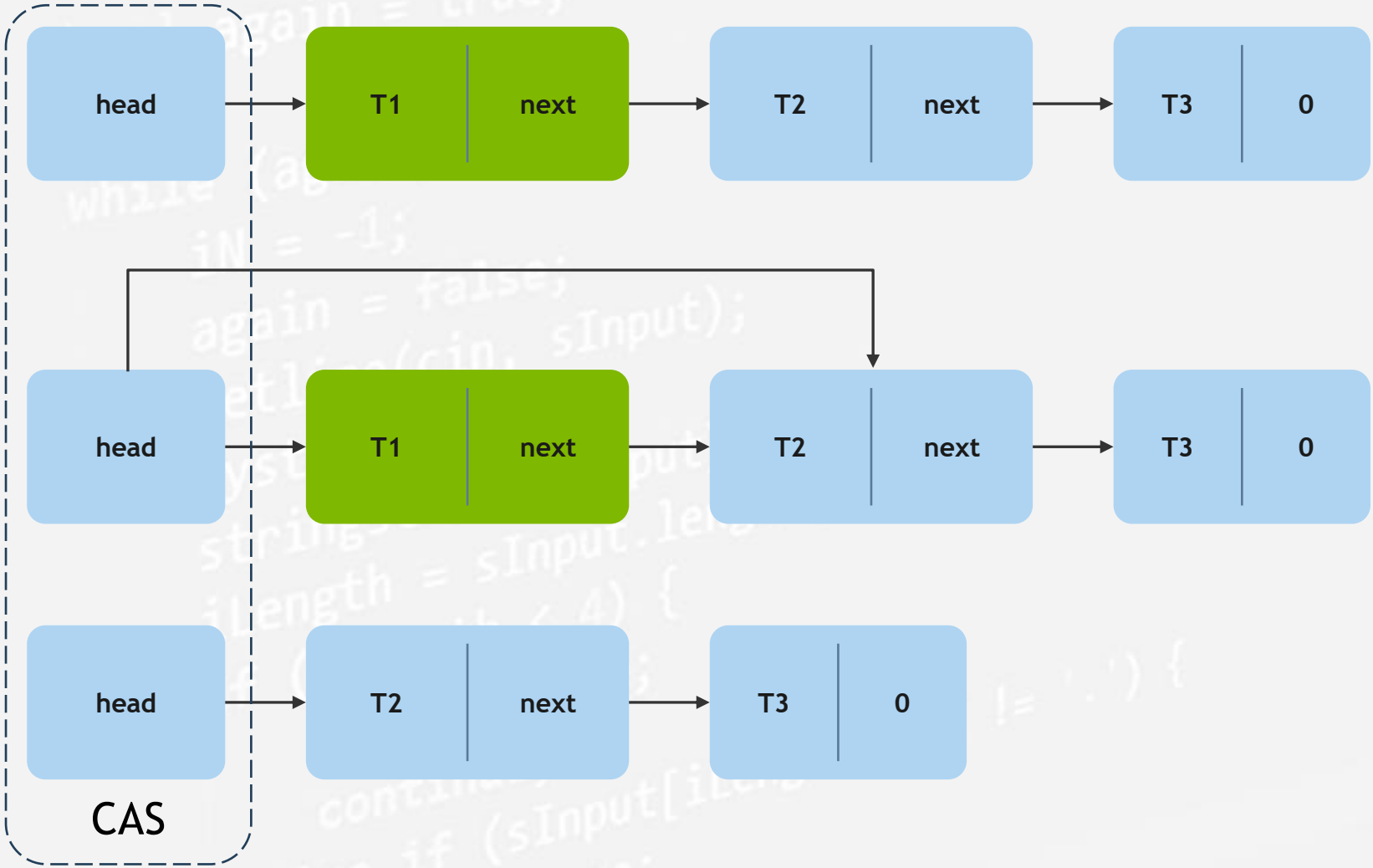
Thread-Safe Forward List

- Lock-free forward list
  - All pointers are atomic
  - CAS is used to update pointers
  - Lifetime is managed by … TBD

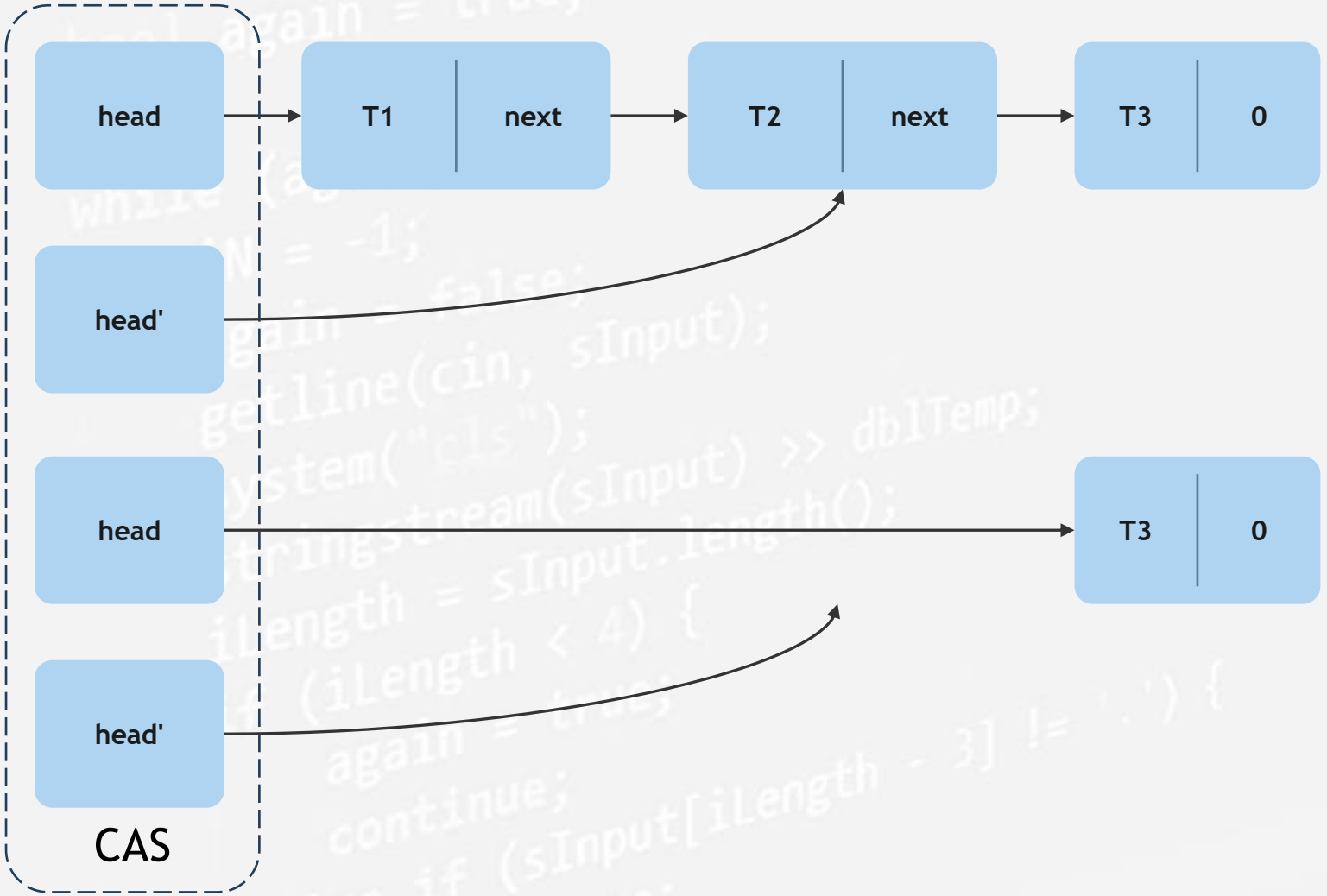
Lock-Free Insertion



Lock-Free Erasure



The ABA Problem

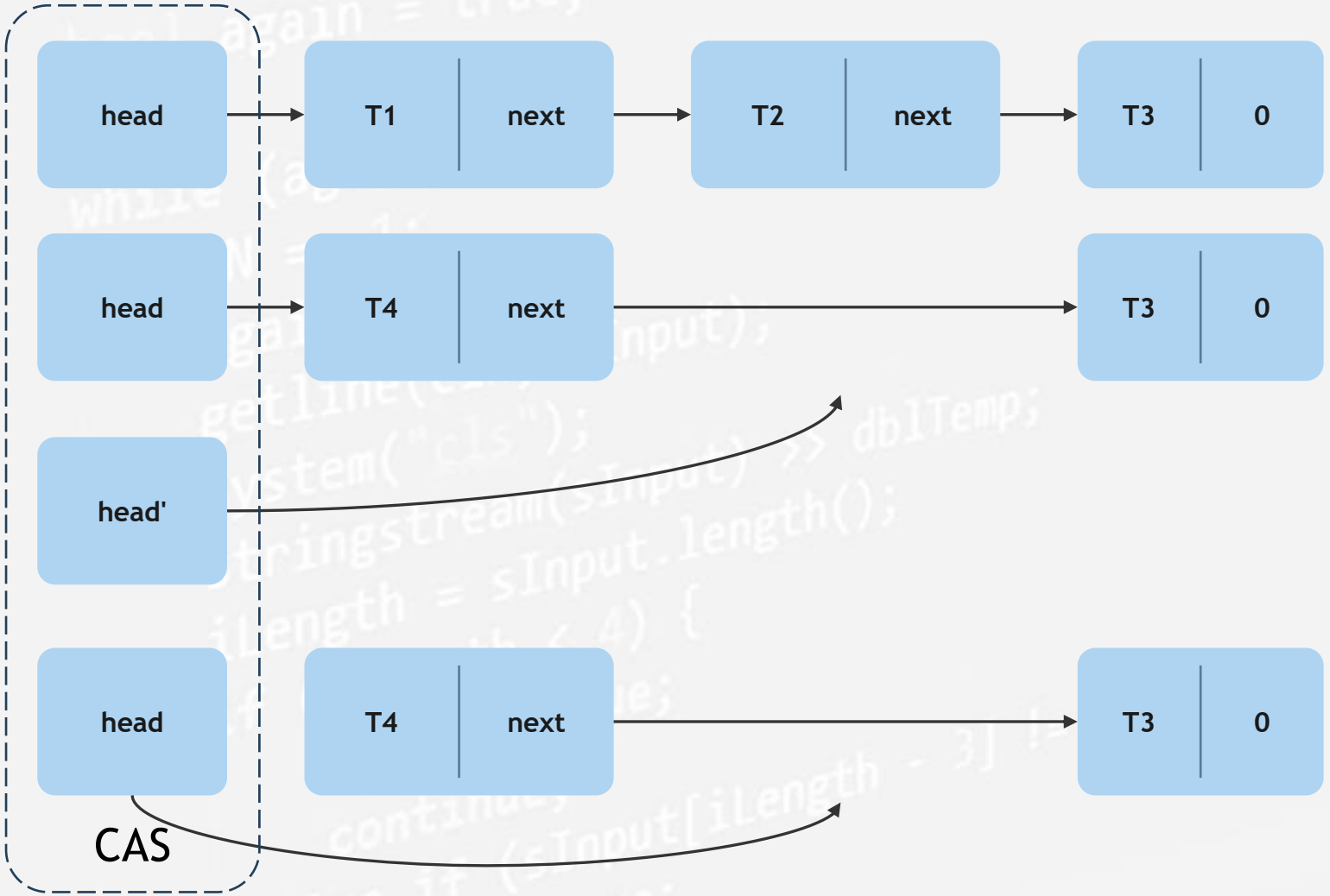


1. Thread 1 to erase T1: read T1.next

2. Thread 2 erases T1 & T2

3. Thread 1 waits, has head'=T1.next

The ABA Problem



1. Thread 1 erases T1

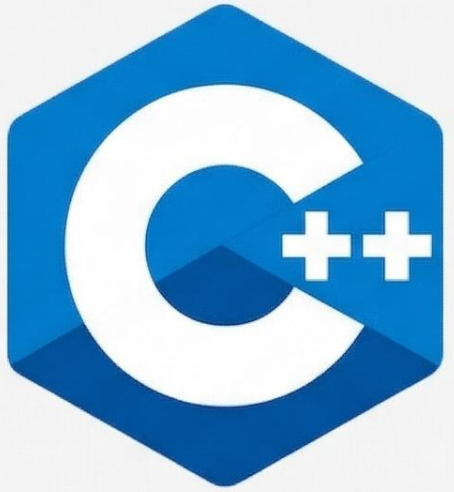
2. Thread 2 inserts T4, gets T1's address

Thread 1 waits, has head'=T1.next

3. Thread T1 CAS succeeds

The ABA Problem

- Address is not a unique identifier if memory can be reused
- Affects all applications of CAS on addresses
  - And any other location reference (slot index, etc)
- Part of the more general problem: lifetime management



Lifetime Management in Concurrent Data Structures

Lifetime Management Is Hard

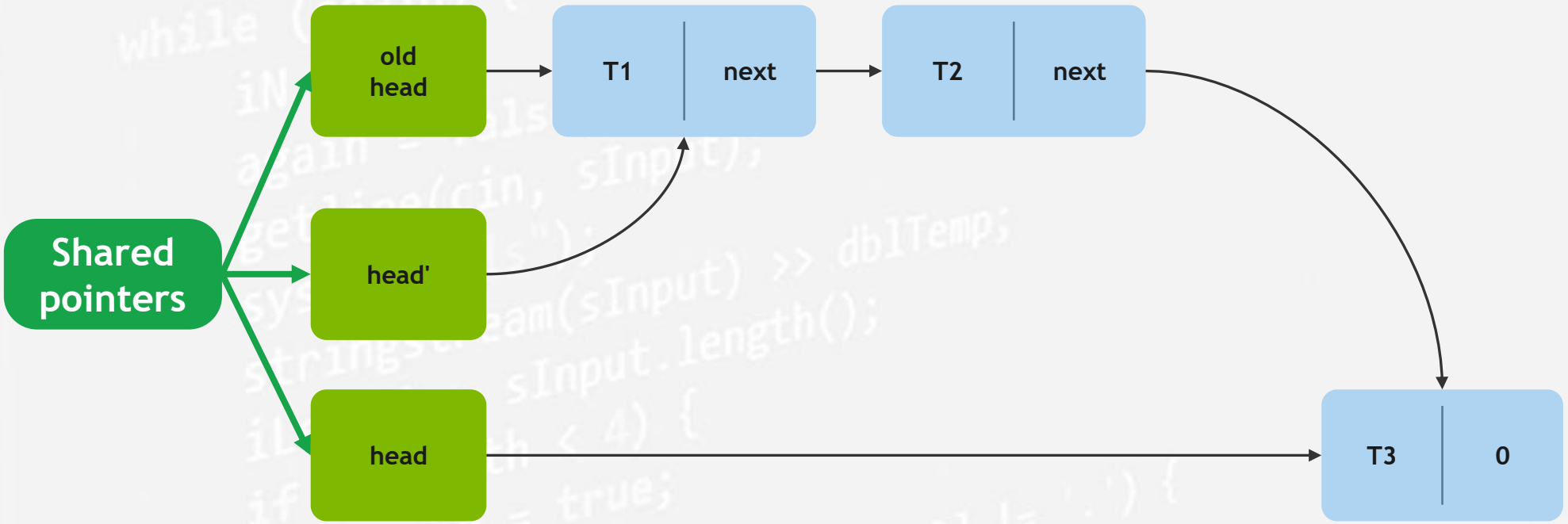
- How to make sure no thread can access objects whose lifetime has ended?

1. Accessing freed memory (use-after-free)
   - Particularly bad since we cannot check object state at all
2. Accessing destructed objects
   - Happens when we try to counter 1
   - Can still read memory, safe for built-in types
3. Accessing object whose identity changed
   - ABA problem
   - Reusing objects – happens along with 2

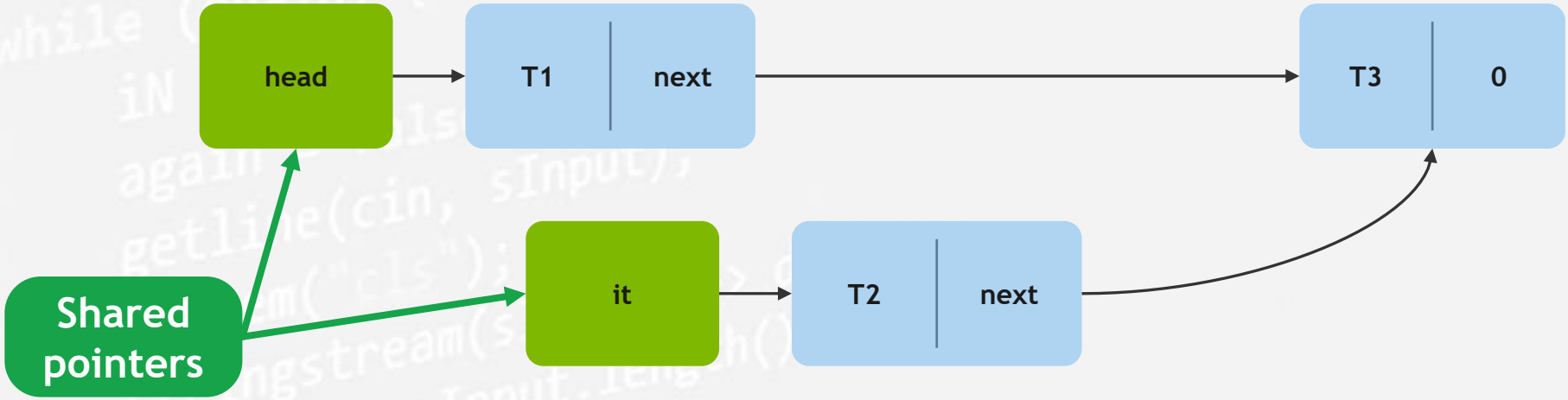
Lifetime Management in Concurrent Data Structires

1. Do not free or reuse memory (for the life of the object)
2. Let the caller handle it – concurrent deque does this
3. Recreate objects instead of deleting them
   - Usually relies on atomic integer flags or counters to avoid accessing deleted objects or misidentification
4. Arrange so nothing is destroyed when it can be observed
   - Shared pointers
   - Hazard pointers
   - Read-Copy-Update (RCU)

Lock-Free List with Shared Pointers



Lock-Free List Iterators



Atomic Shared Pointers

- Two threads can operate on the same node:
- The shared pointer object can be modified concurrently
  - std::shared_ptr is safe when two threads modify separate pointers that point to the same object (reference count is atomic)
- Lock-free list needs atomic shared pointers:
  1. std::atomic<std::shared_ptr> - not lock-free
  2. Lock-free shared pointer using hazard pointers (Daniel Anderson)
  3. Intrusive shared pointer – keeps state in the target object
     - Spinlock using a bit in the state
     - Lock-free using hazard pointers (C++26, Maged Michael)

Read-Copy-Update

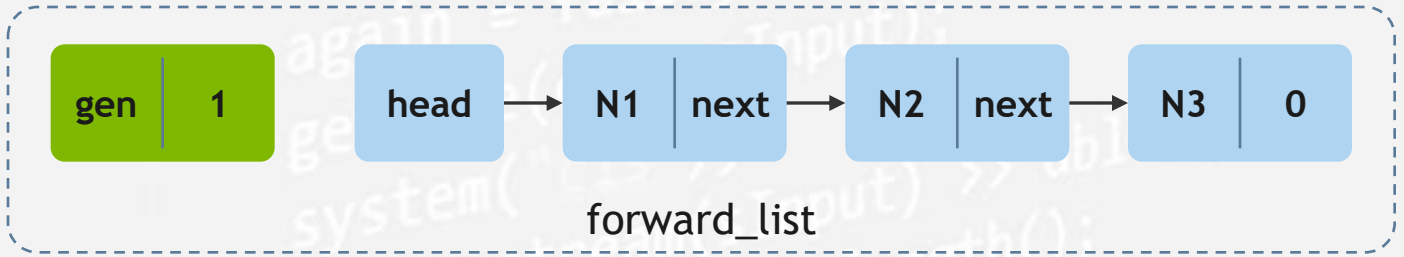
- RCU protocol – deferred reclamation

1. Deleted nodes go to retired lists (not free list!)
   - No ABA problem – memory cannot be reused
   - No lifetime problem – nodes are retired not destroyed
2. Retired nodes are destroyed and transferred to free lists when it is safe to do so
   - No outstanding references from any caller
   - No internal references except retired list maintenance
   - When is it safe to do so?

Read-Copy-Update

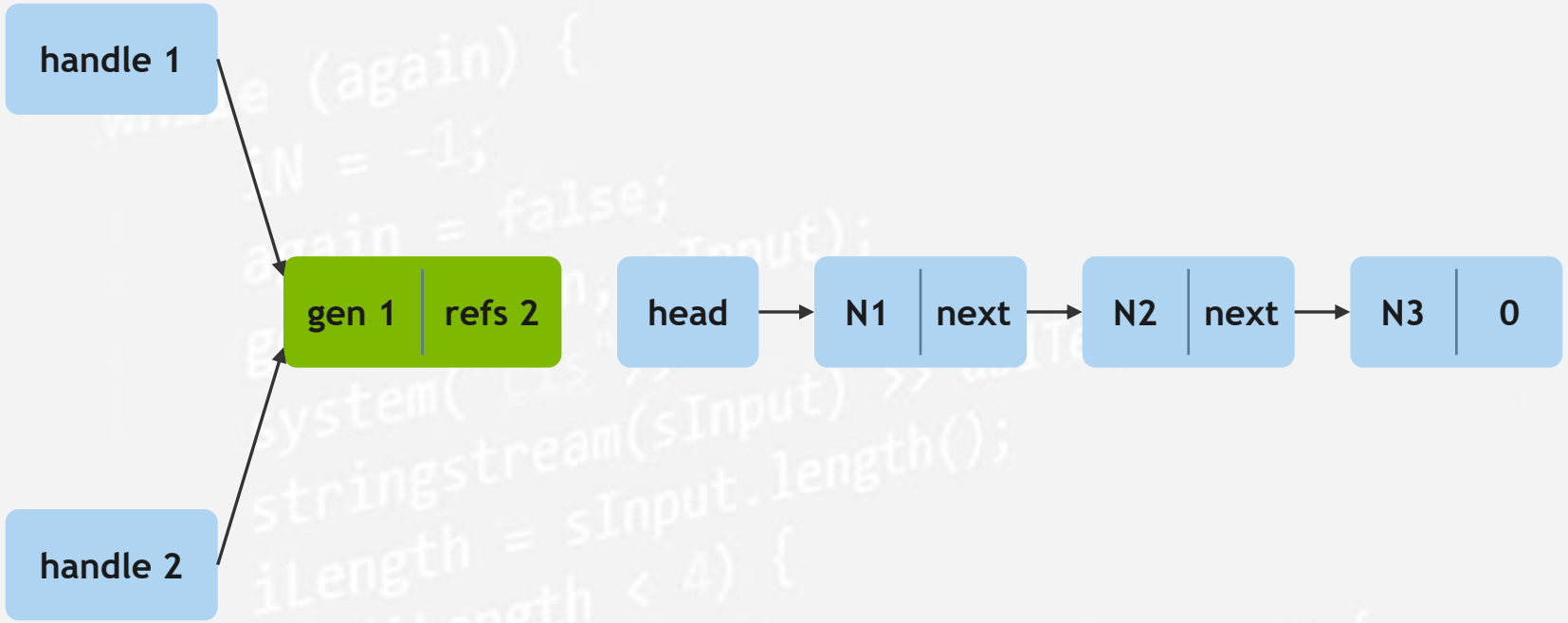
- The data structure (list) maintains a generation number
- ALL operations on the list require a handle
  - Obtained by new_handle() call, opaque type
  - Handles are like shared pointers to generations – reference counting
- Retired nodes carry generation when they were retired
- Generation number advances:
  - No new handles to the old generation can be given out
  - Eventually all old handles will go away
- Callers are not allowed to retain any references without the handle!
- Retired nodes of the oldest unreferenced generation are safe to free

RCU: Generation Number



List maintains generation = 1

RCU: Handles

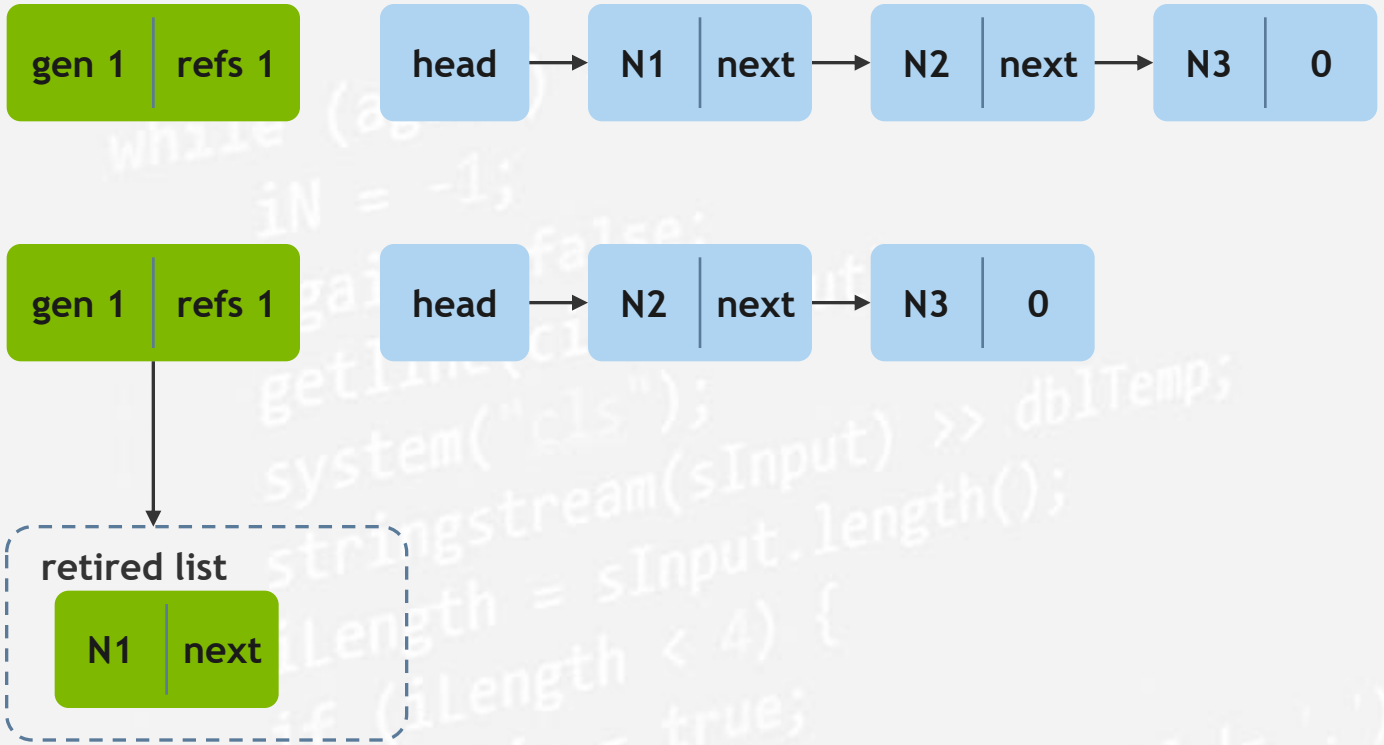


1. new_handle() gives Thread 1 a handle

2. Handles refer to the generation

3. Thread 2 gets its own handle

RCU: Retired Nodes

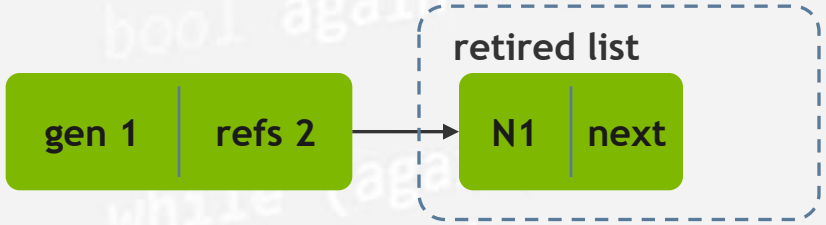


1. List at generation 1

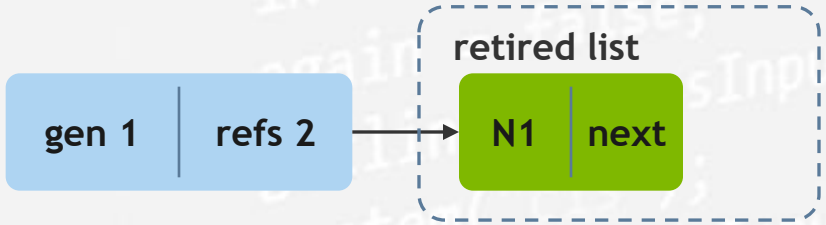
2. Erase N1: unlink it from the list

3. N1 hangs off the status block of gen 1 on its retired list

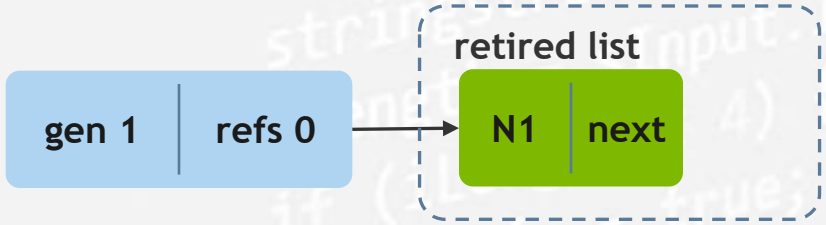
RCU: Generation Advances



1. gen 1 is current; two handles hold it

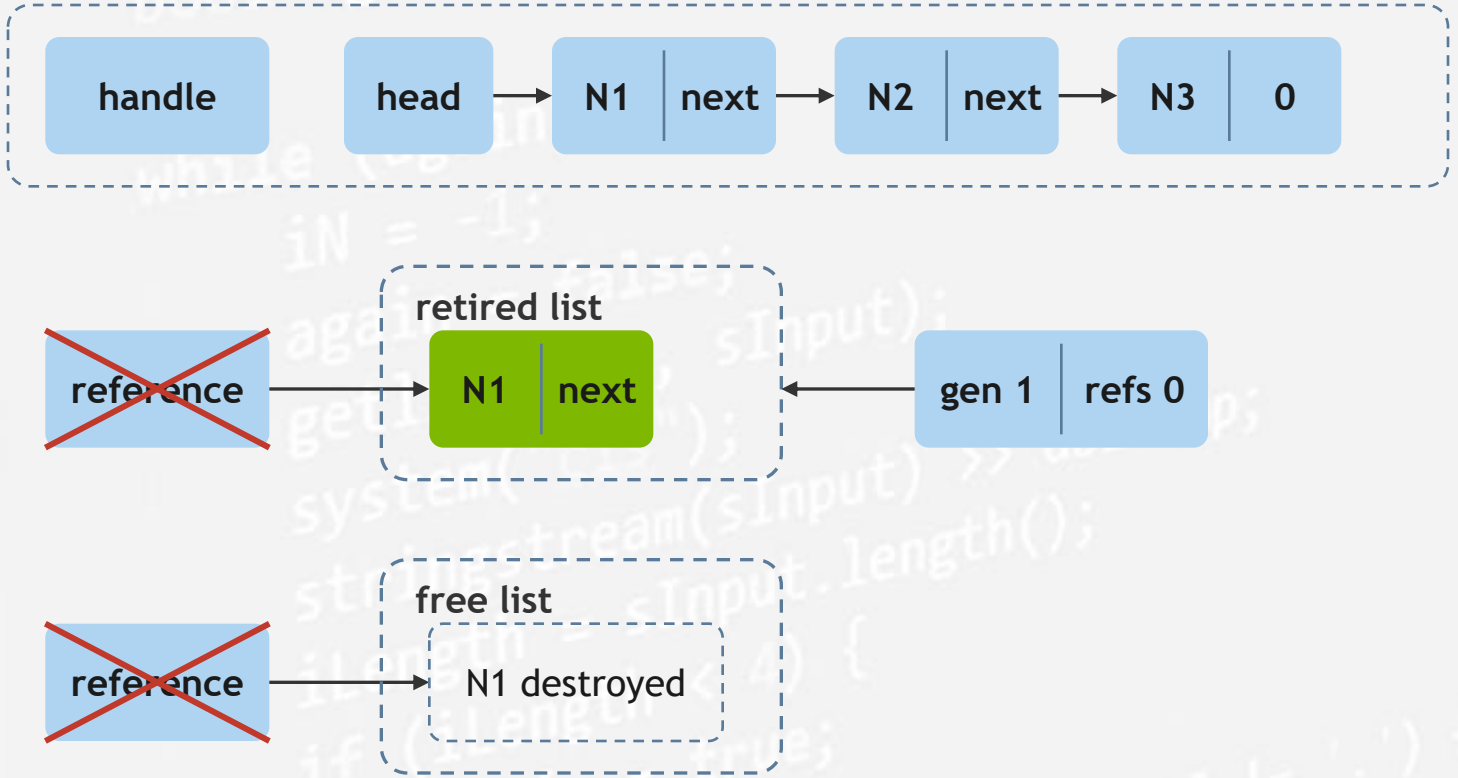


2. Generation advances: new handles get gen 2



3. Old handles go away: gen 1 has no refs

RCU: No References Without a Handle

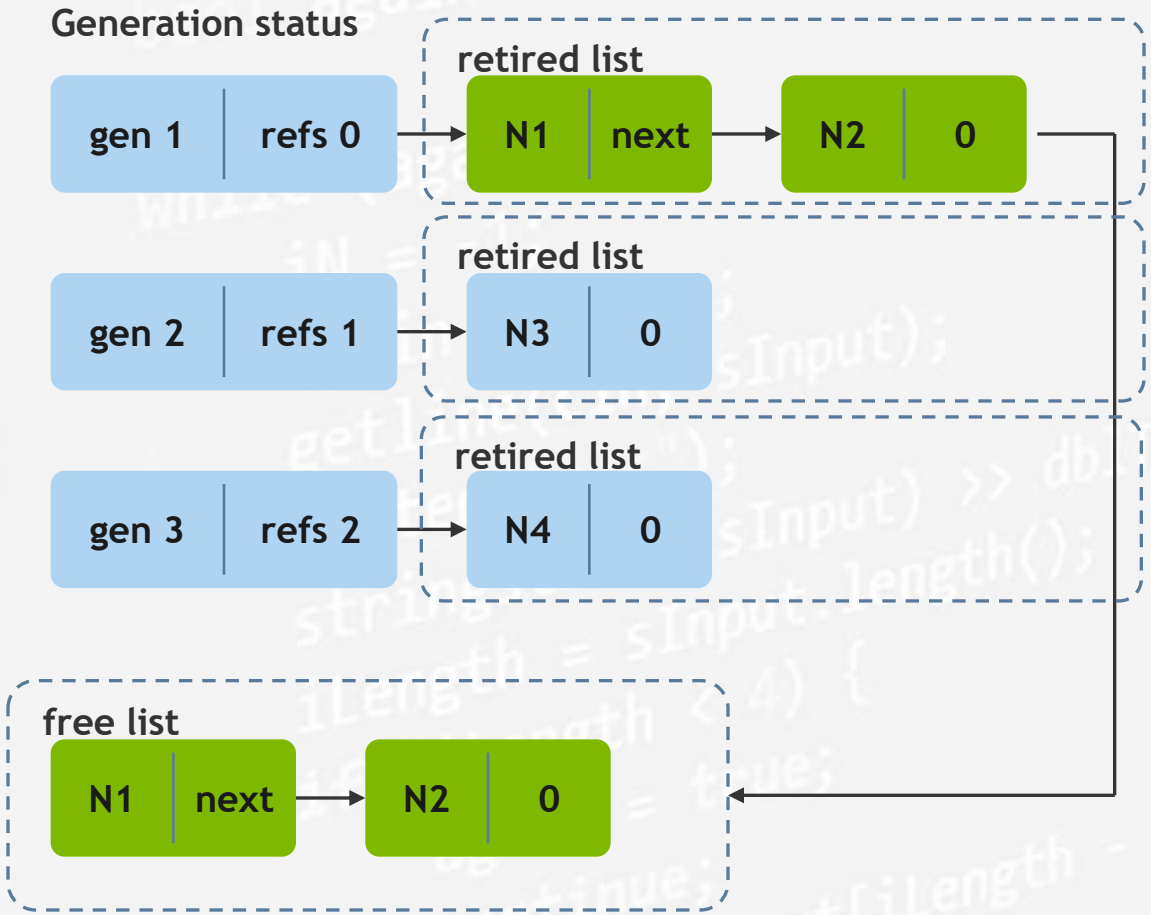


1. Reads under a handle: nodes stay valid

2. Handle released: gen 1 has refs 0; N1 stays on its retired list

3. N1 moves to the free list

RCU: Freeing Retired Nodes



1. gen 1 has refs 0: free its retired list

2. gen 2 still has a handle: keep

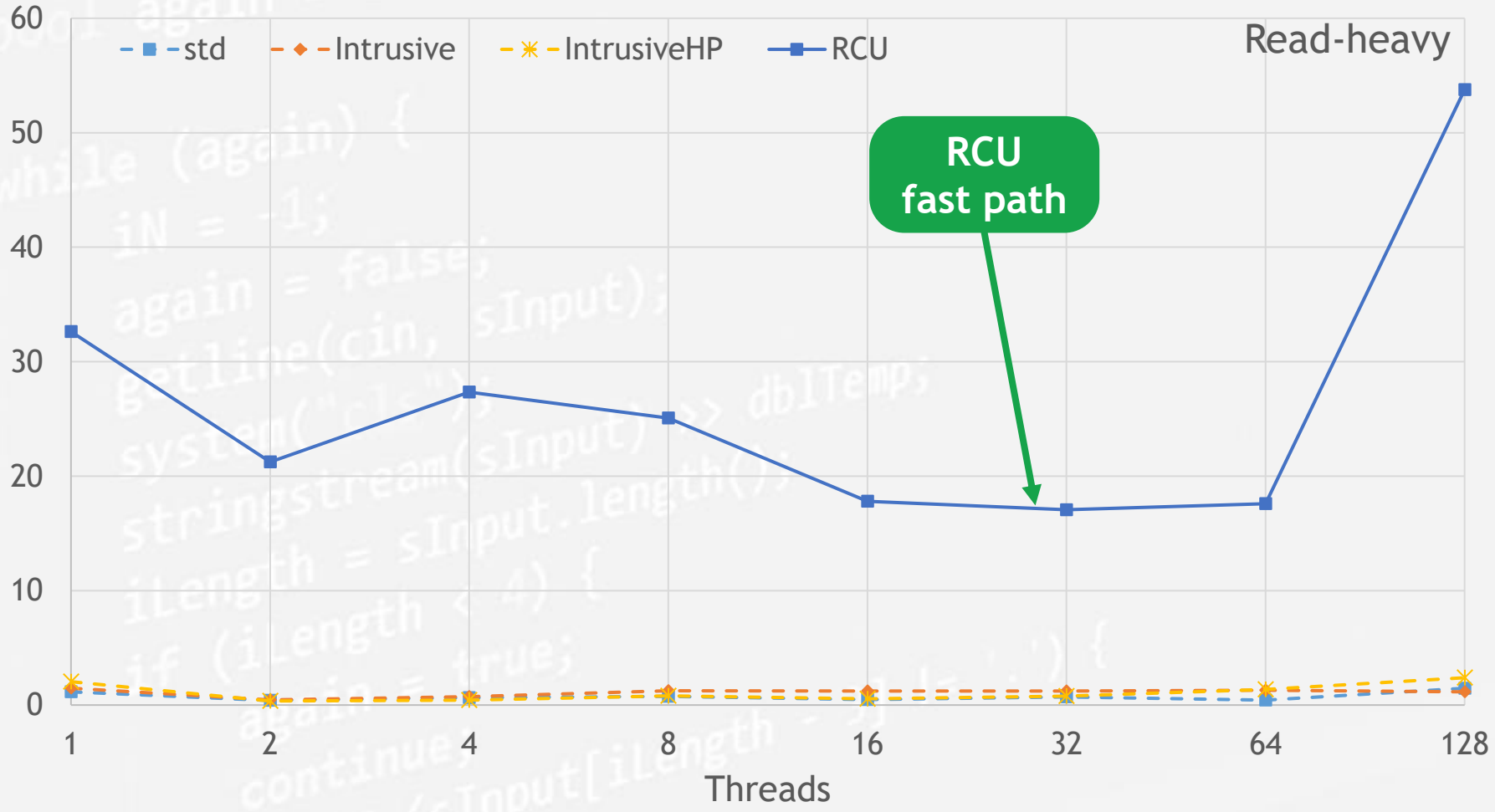
3. Newer generations wait

4. N1 and N2 move to the free list

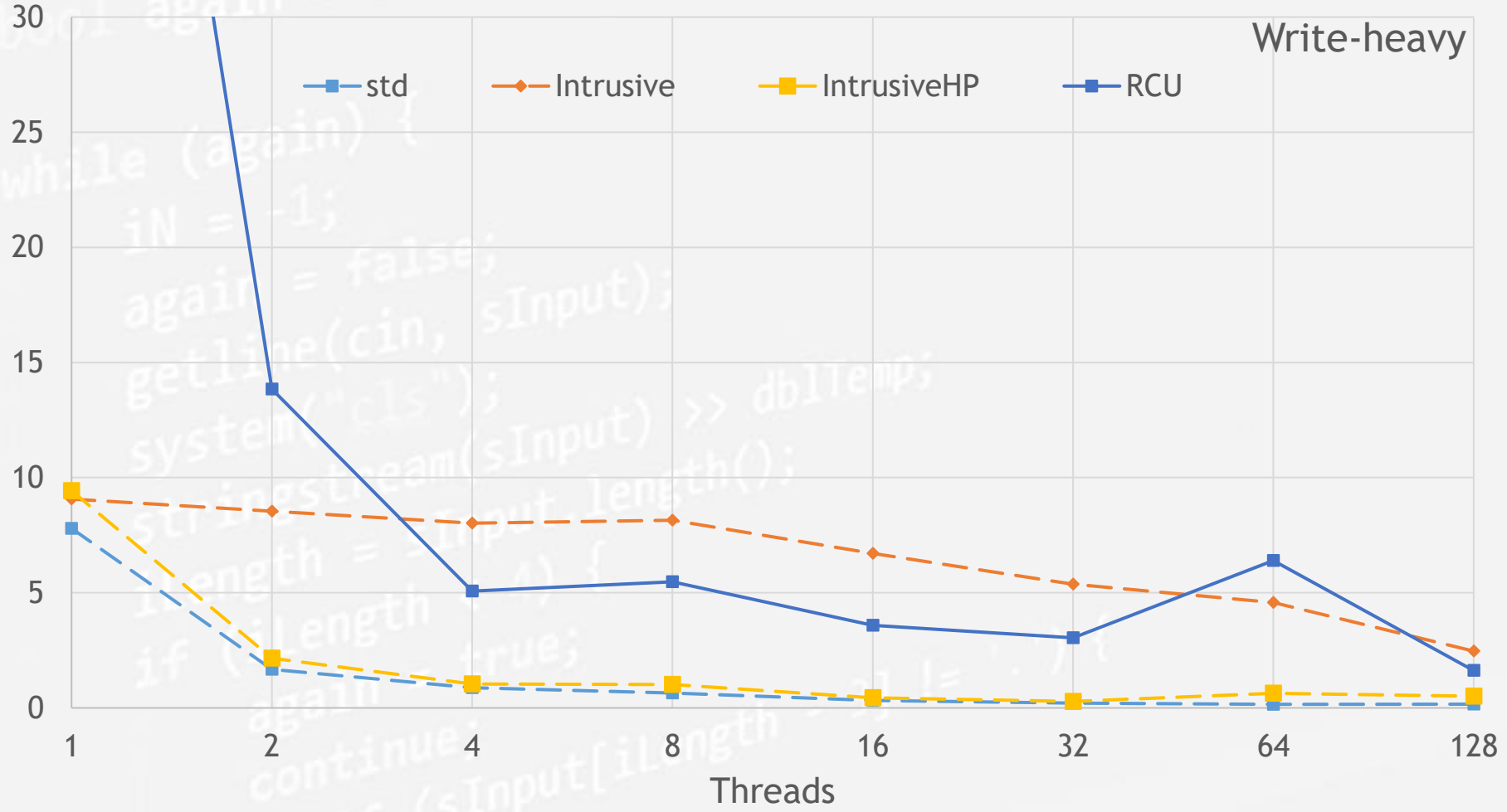
Read-Copy-Update

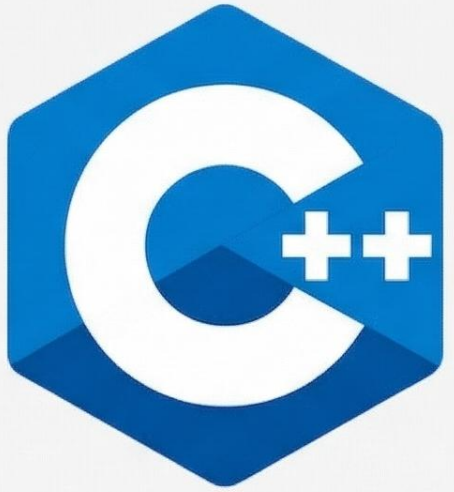
- No shared pointers!
- Fast path – reading the list – is very fast
  - This is a general feature of RCU implementations
- Destructors are delayed until retired nodes are freed
  - Erasing a node never guaranteed its destruction
  - In practice, most of the time it happens at erasure or soon after
- Higher peak memory: retired nodes are kept
  - How much extra memory is used depends on the reclamation policy and how long readers keep references

RCU Performance (Read-Heavy)



RCU Performance (Write-Heavy)



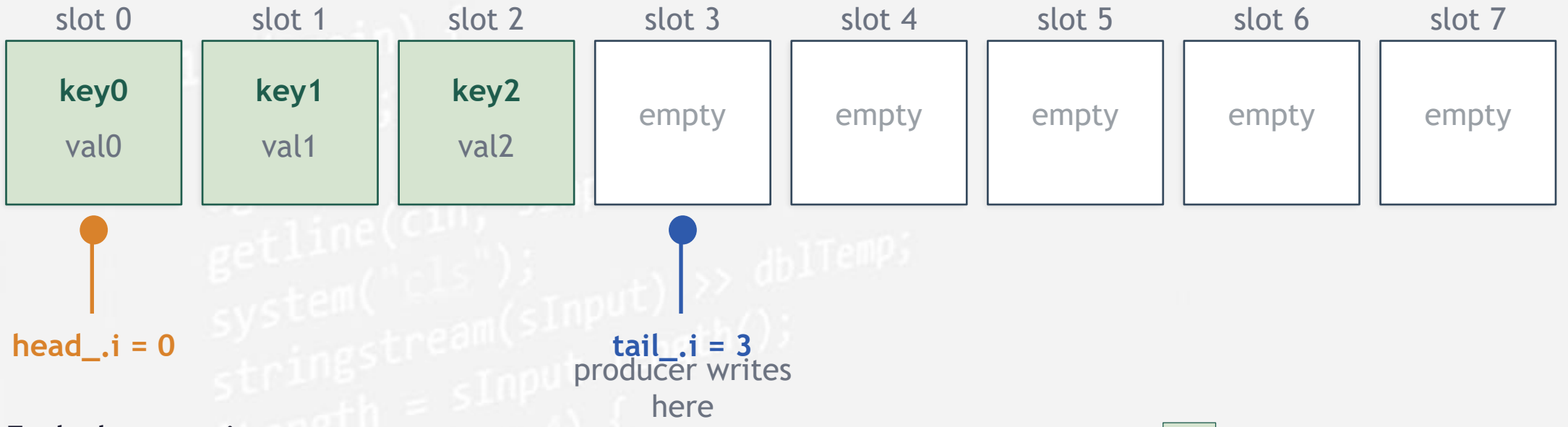


Multi-Producer Multi-Consumer Queue

Concurrent MPMC Queue

- Not-quite-lock-free MPMC queue:
  - spinlock → atomic handoff
- Power-of-2 ring buffer with head/tail indices
- Separate spinlocks for producers (tail) and consumers (head)
  - Spinlock held only during slot claim — released before expensive work
- Per-slot atomics (key + busy flag) for lock-free commit
- Not lock-free – preemption is possible
  - Latency tail needs to be measured
- Requires “reserved” value for keys (e.g., nullptr for T*)

Ring Buffer (Standard Practice)



Each slot contains:

atomic<Key> key	Non-zero = occupied, zero = empty
atomic<int> busy (if Value)	1 = mid-commit, 0 = stable
Value value (optional)	Constructed/destructed in place

Occupied

Empty

Two-Domain Synchronization

Producers (push) — spinlock domain

High contention: every push takes this lock

tail_l
spinlock

tail_i
index, written under lock

Consumers (pop) — spinlock domain

High contention: every pop takes this lock

head_l
spinlock

head_i
index, written under lock

l.release() — spinlock released here; everything below runs with no lock held

Per-slot atomic domain

Low contention: each slot has its own key / busy atomics

Construct value
placement new

key.store(k, release)
→ busy.store(0)

Move value out
destruct in place

key.store(0, release)
→ busy.store(0)

⚠ **Spinlock domains live on separate cache lines — push and pop never contend on the same lock**

Push Operation: The Lock Handoff

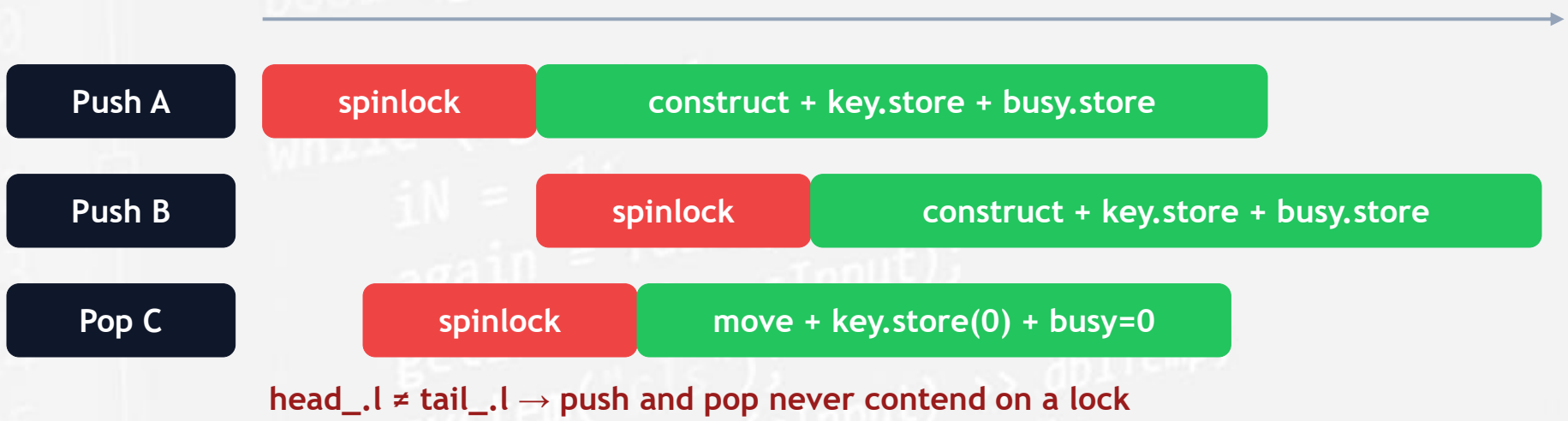
1 Acquire tail lock	tail_.l.lock()
2 Check key==0 && busy==0	Verify slot is empty and not mid-commit (with retries)
3 Set busy=1, ++tail_.i	Claim slot, advance index
4 Release spinlock	l.release() — other pushers unblocked
5 Construct value	placement new — can be expensive, no lock
6 key.store(k, release)	Release fence: value visible before key
7 busy.store(0, release)	Slot fully committed, consumers can pop

Spinlock domain
lock held: steps 1-4
~4 cheap operations

Atomic domain
no lock: steps 5-7
expensive work

⚠ **Lock handoff: release the spinlock before constructing — other pushers proceed while this value is still being built**

Contention Profile



A and B work in parallel on different slots — B waits only for A's ~4 locked ops

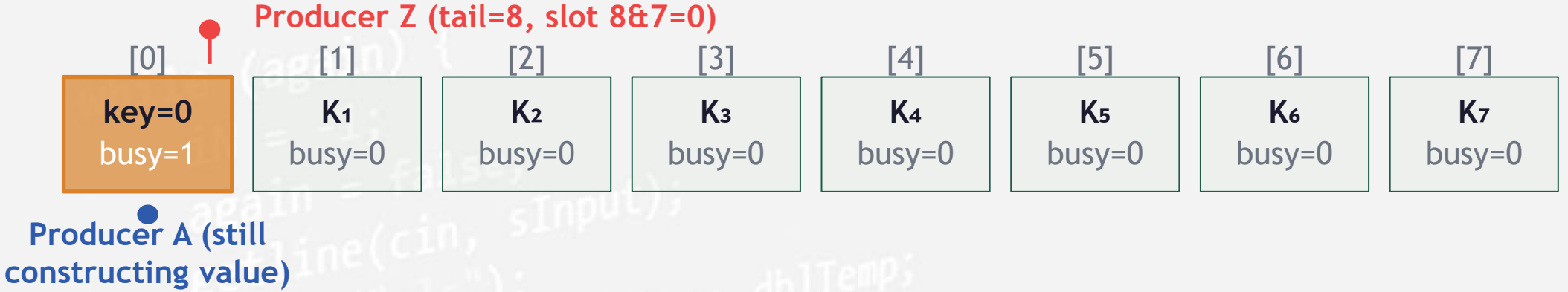
head_.l ≠ tail_.l → push and pop never contend on a lock

Spinlock domain (high contention)
~3–4 operations: load key, load busy, store busy, increment index

Atomic domain (low contention)
Value construction, key store, busy clear — all on separate slots, rarely contend

⚠ Communicating across lock domains: push key.store(release) → pop key.load(acquire)

Producer Wraparound Race

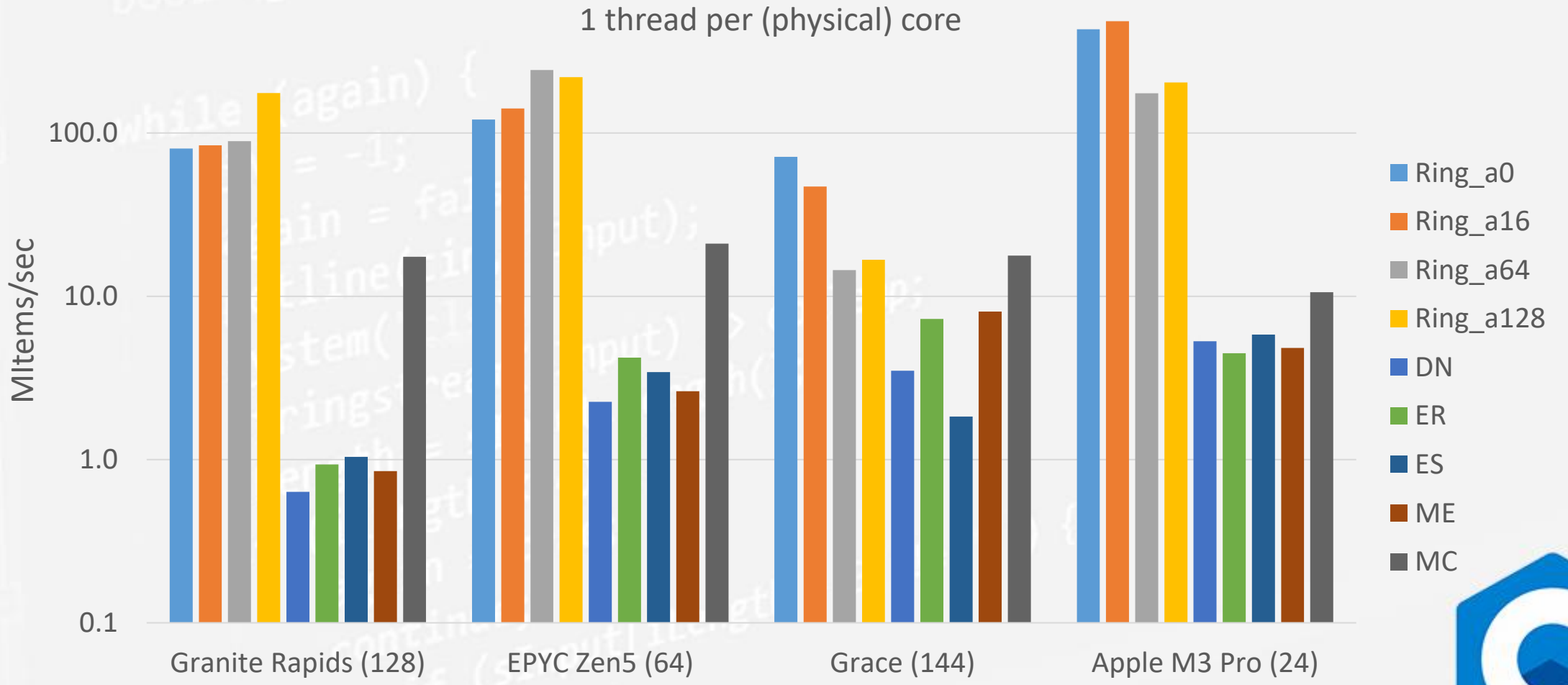


1. Producer A claims slot 0: busy=1, tail++ → releases lock
2. Producers B–H push to slots 1–7, all commit successfully
3. Producer Z acquires lock. tail_.i=8, slot = 8 & 0x7 = 0
4. Z sees key==0 (A hasn't stored key yet!) — looks empty!
5. ✓ But busy==1 — Z knows a commit is in flight, retries via NTRY

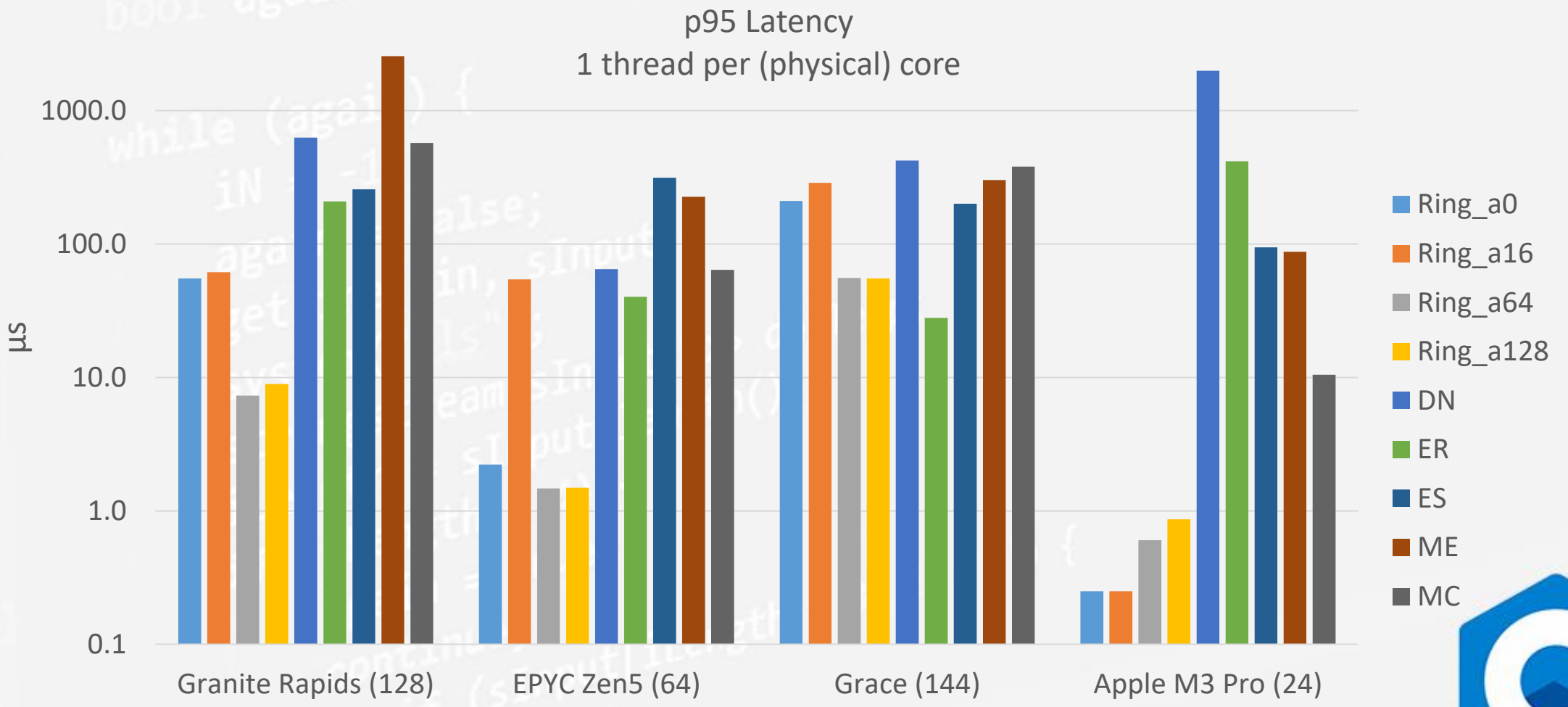
Performance Must Be Measured

- Ring queue (multiple slot alignments)
- https://github.com/DNedic/lockfree
  - Djordje Nedić
- https://github.com/rigtorp/MPMCQueue
  - Erik Rigtorp
- https://github.com/erez-strauss/lockfree_mpmc_queue
  - Erez Strauss
- https://github.com/max0x7ba/atomic_queue
  - Maxim Egorushkin
- https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm
  - Cameron Desrochers

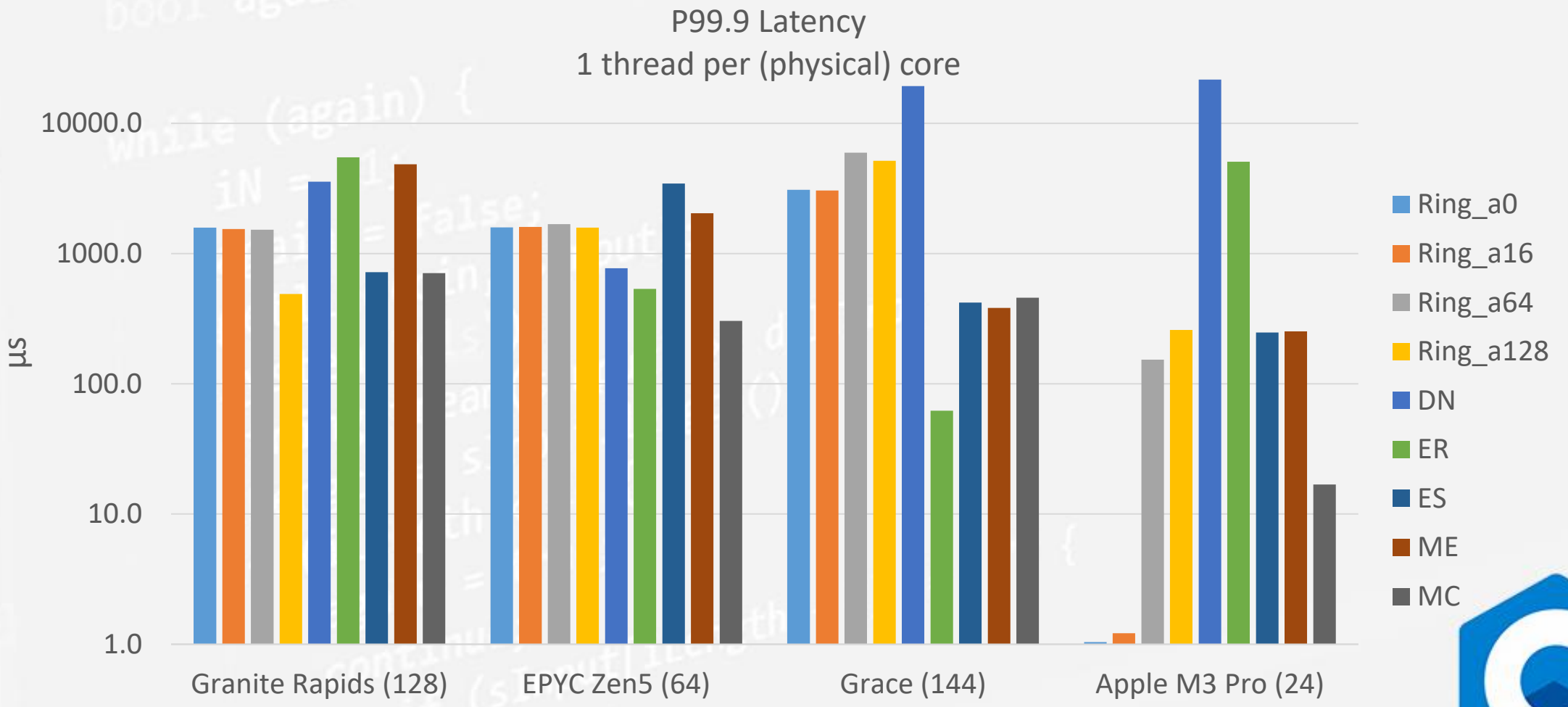
Throughput (Loaded Queue)



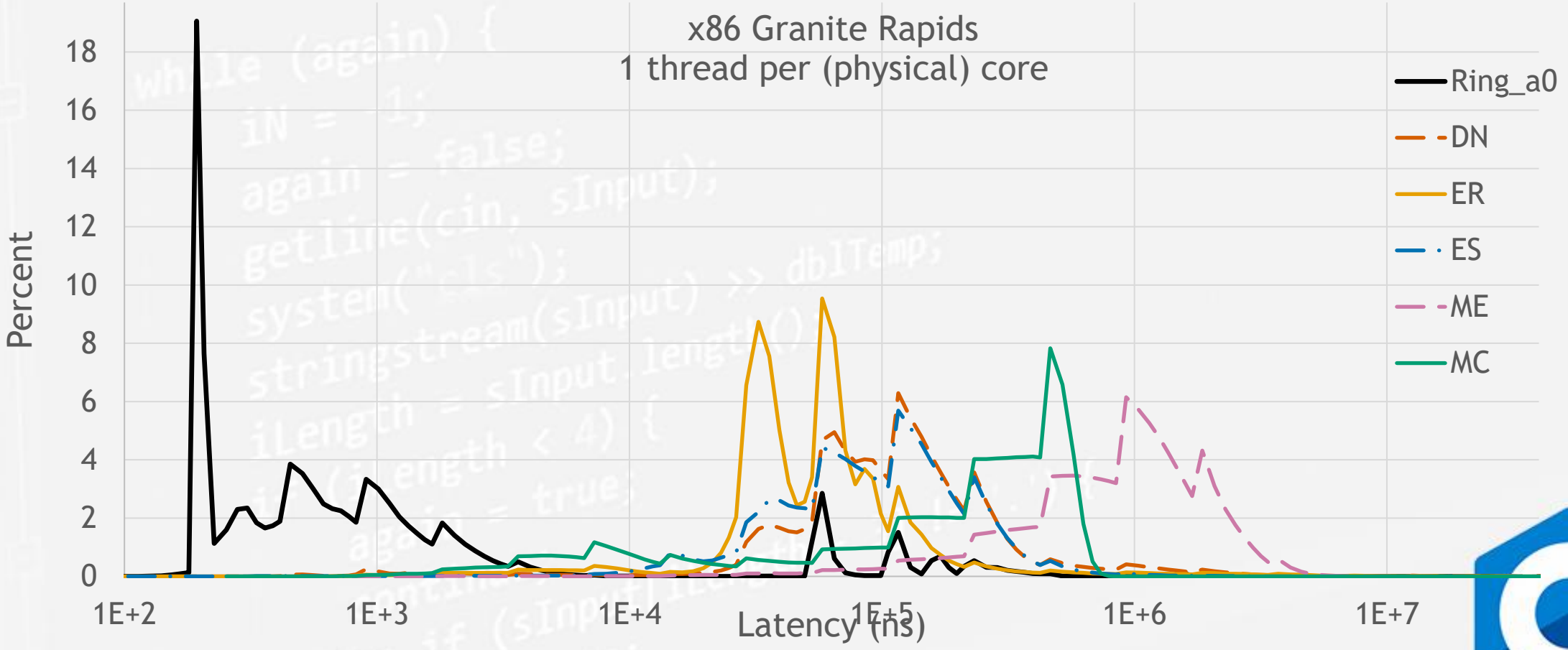
Latency (Near-Empty Queue)



Latency (Near-Empty Queue)



Latency Distribution



Latency Tail Summary

- Ring queue – lock-based, subject to preemption
- ES queue – fully lock-free, bounded delays

Percentile	R0 (ring, lock-based)	ES (lock-free)
p50	0.51 µs	77.8 µs
p90	54.7 µs	185 µs
p95	117 µs	228 µs
p99	300 µs	324 µs
p99.9	1.57 ms	0.51 ms
max	46 ms	9.3 ms

R0 is 3× better

ES is 3× better

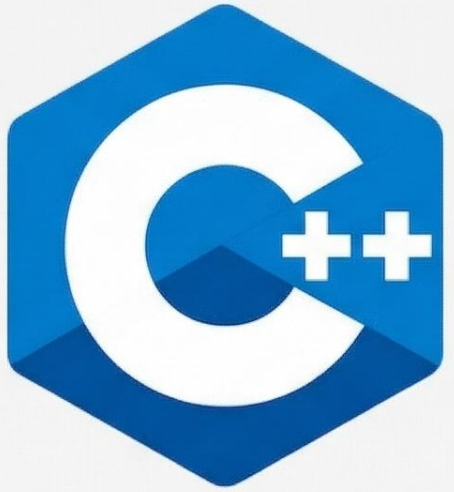
ES is >5× better

Grows with time

Bounded

Benchmark Conclusions

- Design based on the new model appears to work…
  - Spinlocks for high contention
  - Lock-free for low contention
  - Combine locks and atomics for minimum locking
- Preemption is real – lock-based queue has worse outliers
  - At 99.9% latency and beyond – most latencies are better
- Conclusions hold for wide range of hardware
  - x86 and ARM, servers and desktops
- Optimal tuning differs on different hardware:
  - alignment, padding, retries



Lessons Learned

Main Points

- Locks work better at high contention
  - backoff is essential for high-contention writes
  - Read with retries is critical – need to wait for ACKs on RFO
  - Hidden cost of the lock is its release
- Locks can work better with atomics
  - DCLP and other “lock-free style” techniques minimize critical section
- Lock-free works better at low contention
  - Cost of the lock doesn’t matter – disruption of execution flow matters
  - Implicit synchronization: lock variable and guarded variable
  - Complex order restrictions disrupt CPU pipeline

https://github.com/fpikus/ConcurrentCpp

1. **Spinlocks for high contention**
   Slow – but faster than the alternative
2. **Lock-free for (simple) low contention**
   Atomics win when the race is rare and the update is a single CAS
3. **Atomic handoff across domains (DCLP)**
   Peek the atomic outside the lock; take the lock only when it pays
4. **Locks + atomics = less locking**
   Hold the lock briefly, publish with release, read with acquire
5. **The hardest part is lifetime management**
   (a) don't (b) shared pointers & hazard pointers (c) RCU

https://github.com/fpikus/ConcurrentCpp