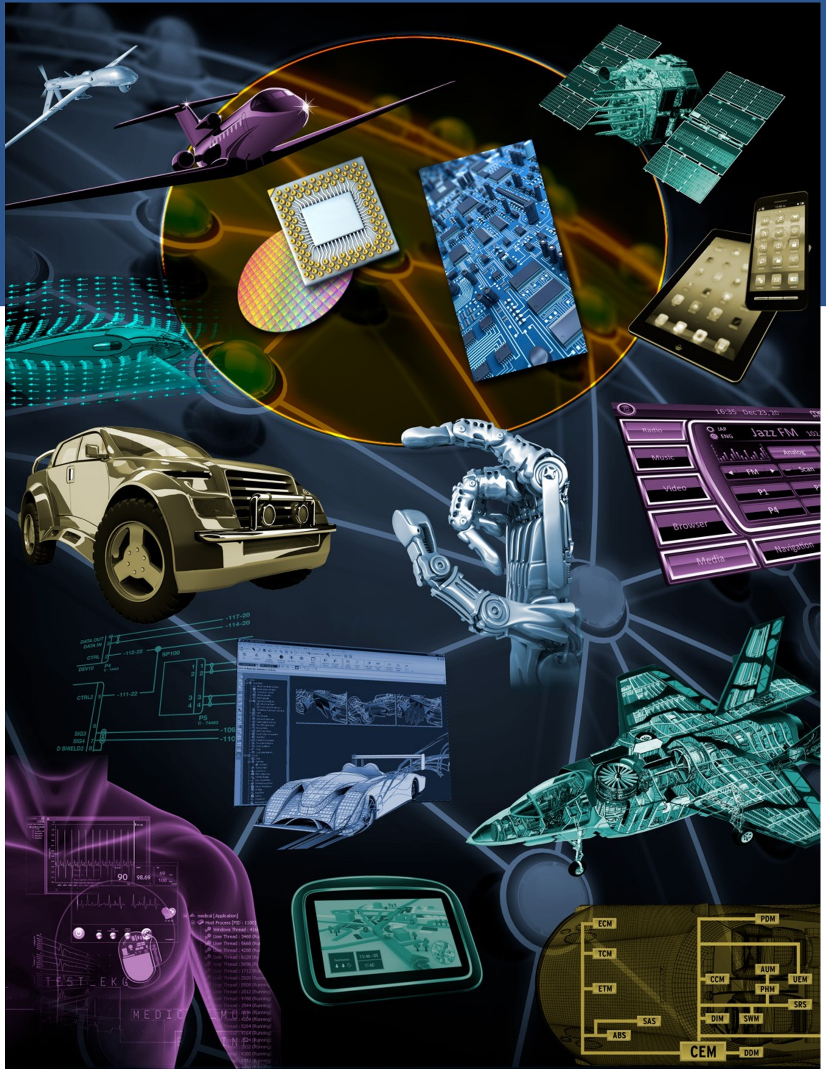


Read, Copy, Update… Then what?

Fedor G Pikus

Chief Scientist

Design2Silicon Division

Semptember 27, 2017

Mentor®

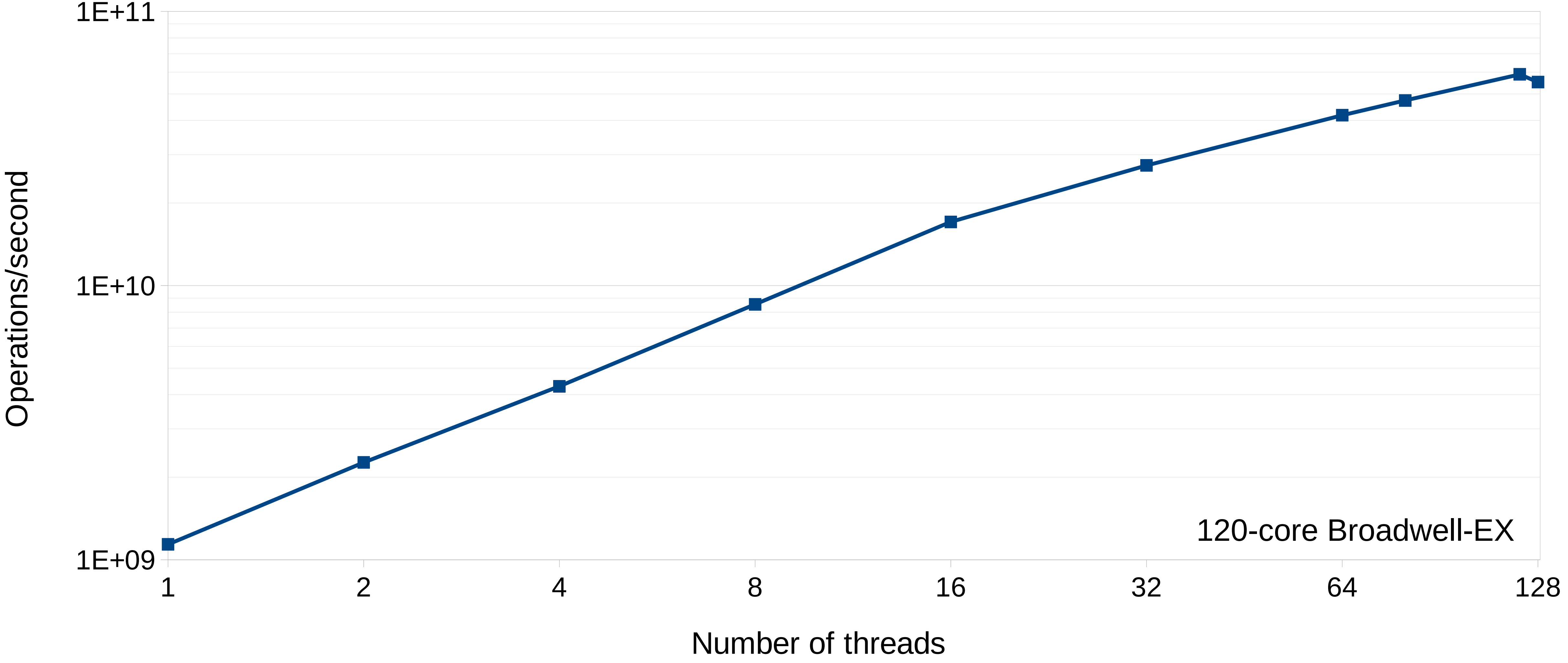
A Siemens Business

Sometimes high performance is easy…

- To get maximum performance you need concurrency
- Simplest parallel problem:
  read-only data+massive computations
  - Results have to be written somewhere…
  - Just lock it if it’s infrequent
  - As long as input data does not change
- Many reader (consumer) threads run entirely independently with no contention or synchronization

Mentor
A Siemens Business

Sometimes high performance is easy: searching read-only list



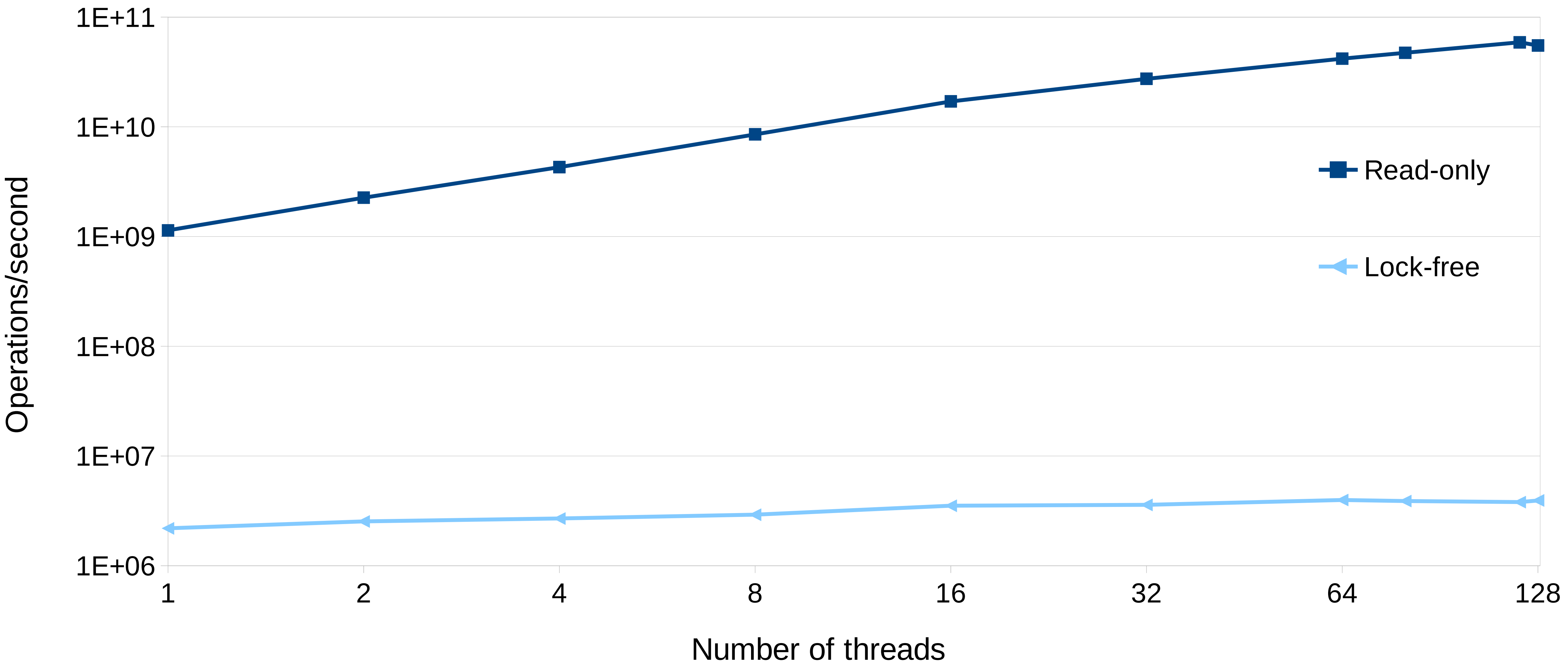
Life is rarely that easy

- Read-only data does happen
  - Often in scientific computations
- More often, data is “almost read-only”:
  - Changes happen but infrequently
- Many reader threads, one writer thread
  - Or write operation is locked, so effectively “one writer thread at a time”
- Reader performance should not be affected (ideally)

Almost read-only data

- Even if writes happen rarely, data can change while a reader is reading it
- Without any synchronization, this is a data race
- We definitely don’t want the readers to wait on each other
- Ideally we don’t want the readers to wait for the writer
- We can’t have the readers block the writer indefinitely
- Ideally we don’t want the writer to wait at all

Searching read-only list, lock-free (list with atomic shared pointers)

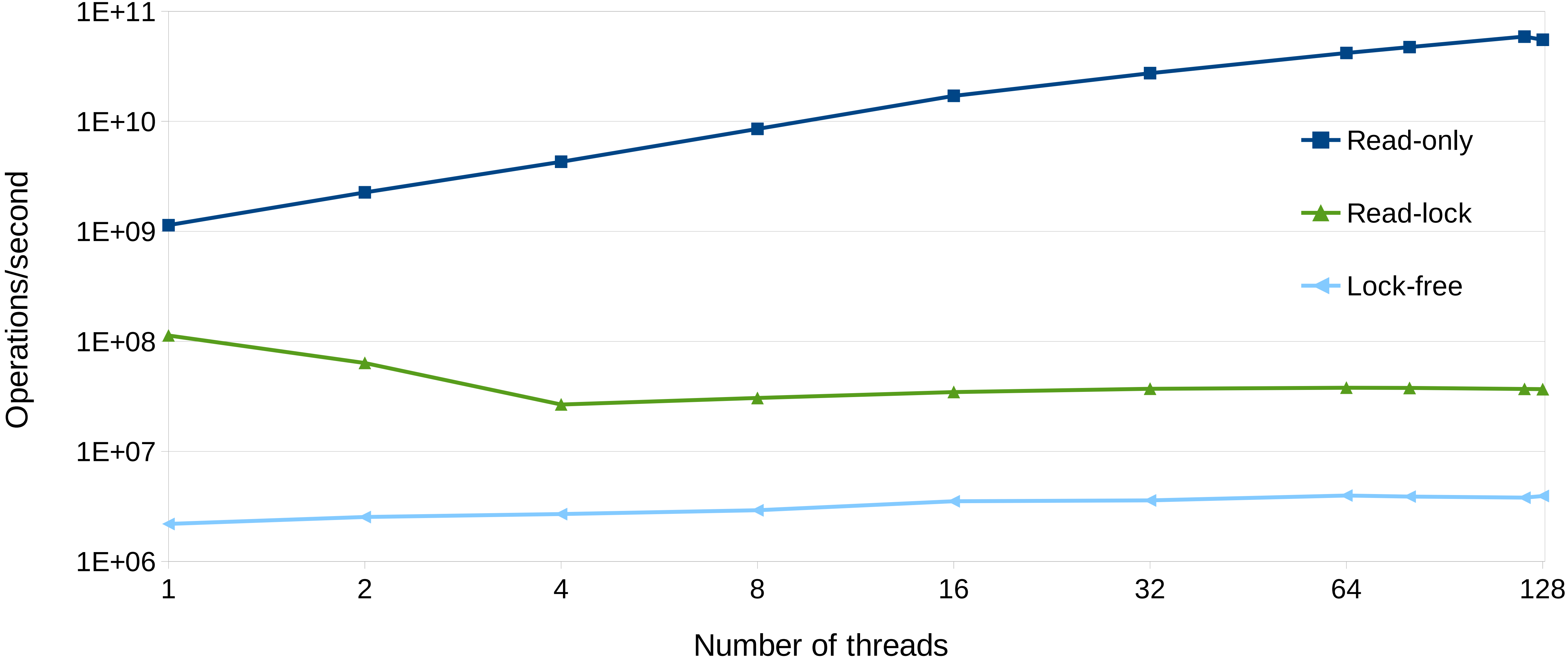


Mentor A Siemens Business

Almost read-only data

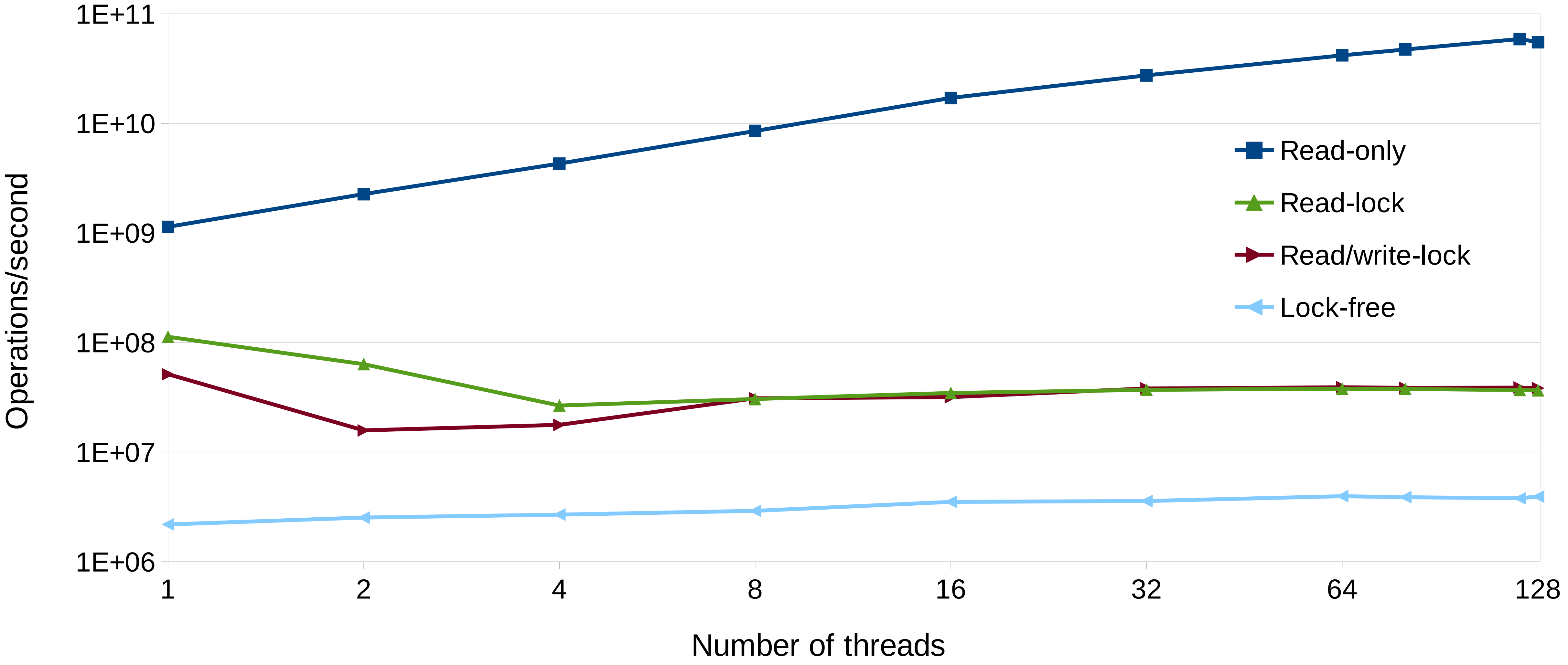
- Lock-free solution is generic, allows for multiple writers, scales well, but has overhead even in read-only case
  - Good solution when you need it
  - Wrong choice when updates are rare
- This is what read-write locks are for...

Searching read-only data with r/w locks (pthread_rwlock_t)



Mentor
A Siemens Business

Searching changing data with r/w locks (pthread_rwlock_t)



Almost read-only data

- Even if writes happen rarely, data can change while a reader is reading it
- Without any synchronization, this is a data race
- There is reader overhead even if no updates actually happen during the test
  - Simply because they could have happened
- This is not fair!

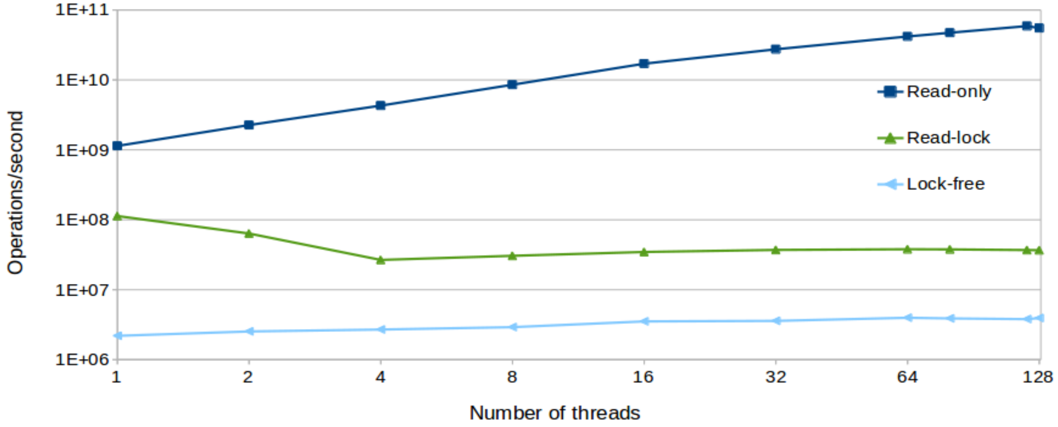
There has to be a better way

Almost read-only data

- Lock-free solution is generic, allows for multiple writers, scales well, but has overhead even in read-only case
  - Good solution when you need it
  - Wrong choice when updates are rare
- This is what read-write locks are for…

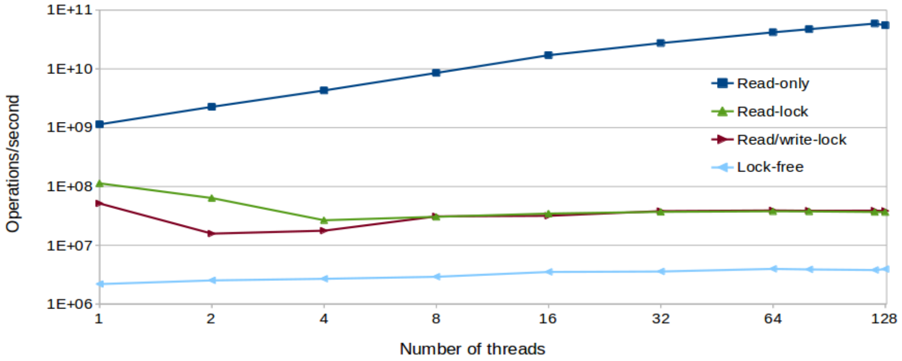
<number>

Searching read-only data with r/w locks (pthread_rwlock_t)



<number>

Searching changing data with r/w locks (pthread_rwlock_t)



<number>

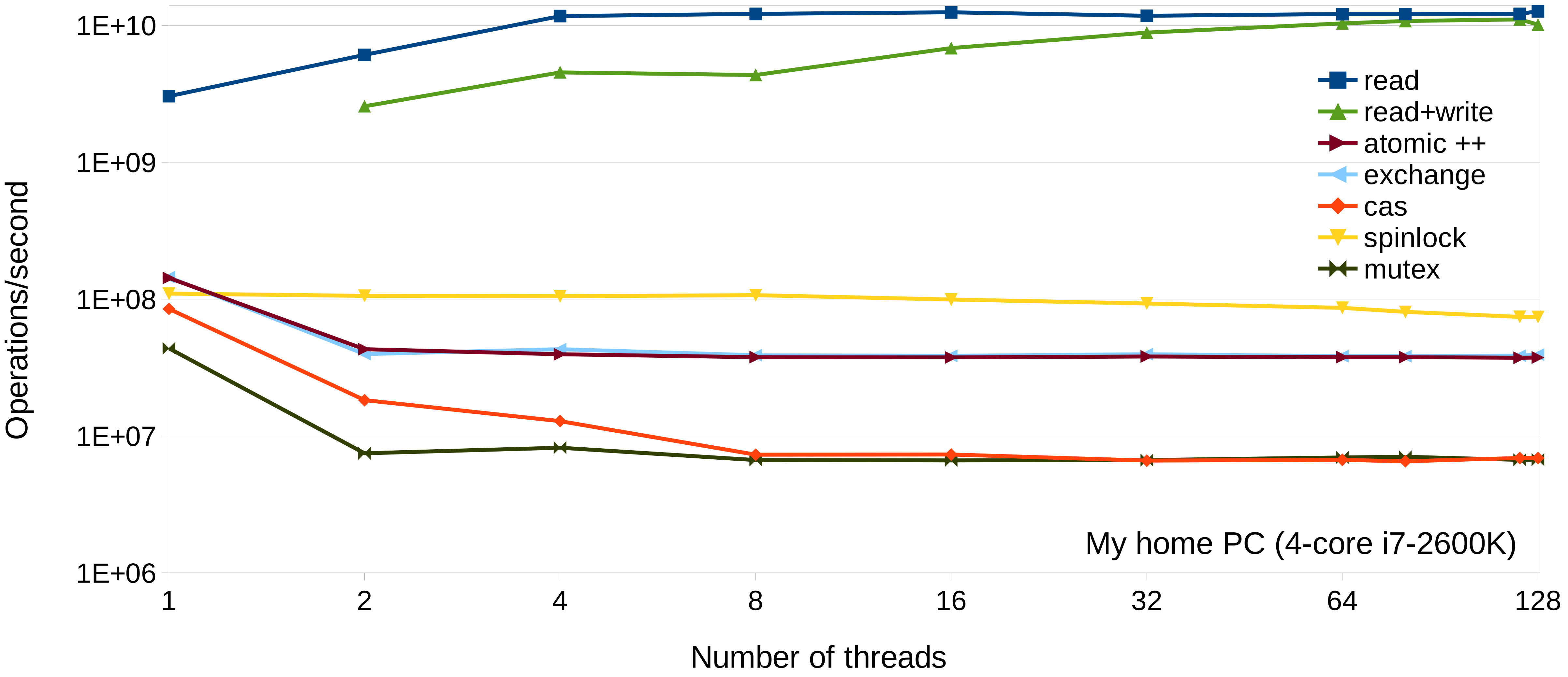
Almost read-only data

- Even if writes happen rarely, data can change while a reader is reading it
- Without any synchronization, this is a data race
- There is reader overhead even if no updates actually happen during the test
  - Simply because they could have happened
- This is not fair!

<number>

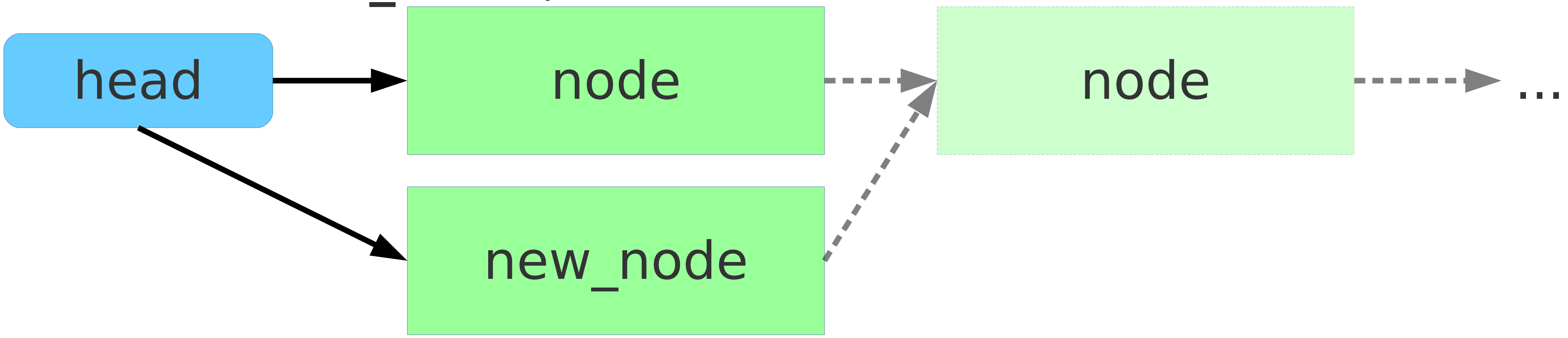
What is the cheapest atomic operation?

What is the cheapest atomic operation?



What can I do with just reads and writes?

- std::atomic<node*> head;
- Reader: node* **p = head**; do_search(p); node or new_node?
- Writer:
  node* new_node = new node(...); // Not visible to readers
  new_node>next = head->next;
  head = new_node;   // Now it's visible



Mentor
A Siemens Business

What can I do with just reads and writes, carefully?

- std::atomic<node*> head;
- Reader:
  node* p = head.**load**(**std::memory_order_acquire**);
  do_search(p);
- Writer:
  node* new_node = new node(…);
  new_node->next =
          head.load(**std::memory_order_relaxed**)->next;
  head.**store**(new_node, **std::memory_order_release**);
- No writer synchronization – only one writer thread!
  – Or use mutex (or something else)

How does the reader work?
How does the writer work?

- Reader – use atomic read, otherwise nothing special:
```
node* p = head.load(std::memory_order_acquire);
do_search(p);
```
- Writer:
```
node* new_node = new node(...);
node* next =
        head.load(std::memory_order_relaxed)->next;
new_node->next = next;
head.store(new_node, std::memory_order_release);
```
Read current data

Copy it to new data

Update current data

- **Read, Copy, Update - RCU**

WARNING: confusing terminology ahead!

- “Read, Copy, Update” is actually a very small part of RCU
- RCU does stand for “Read, Copy, Update”
  - Somewhat of a misnomer
- No standard name for the “RCU” part of RCU
  - “Copy-on-write” in Java, but in C++ we use term COW for something entirely different
- Sometimes called “publishing protocol”

RCU, more generally

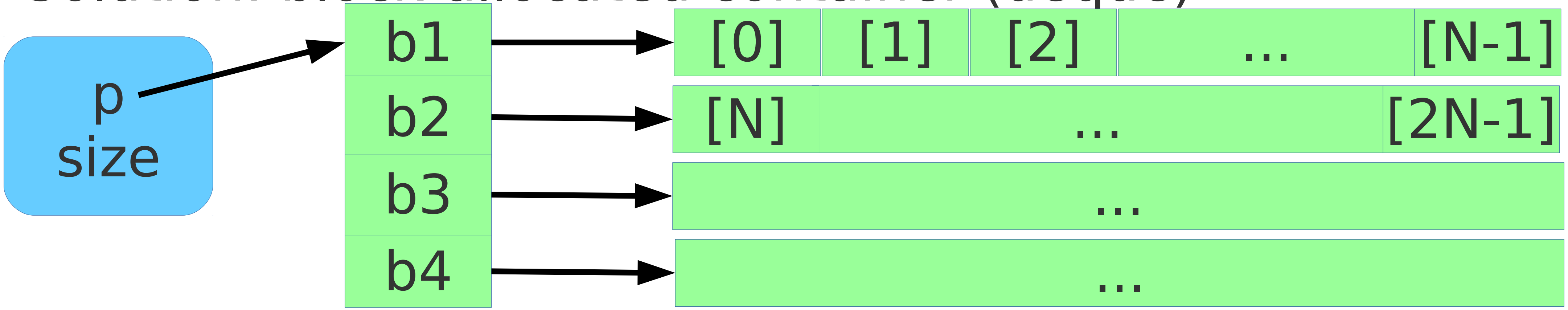
- Data is accessible through “root pointer”
  - Could be index into an array
  - Must be atomic
- Reader: acquire “root pointer” atomically, access data
- Writer: read current data, copy to new data, update new data, and publish it
- Some readers see old data, other readers see new data
  - Normal in concurrent systems

Example: thread-safe “growable array”

- Design an “array-like” container that can grow
  - Resize (grow) can be locked or limited to one thread
  - Array elements that existed before resize remain where they are and can be accessed by any thread, lock-free
- Resize does not invalidate pointers or iterators

First step: how to grow without moving old data?

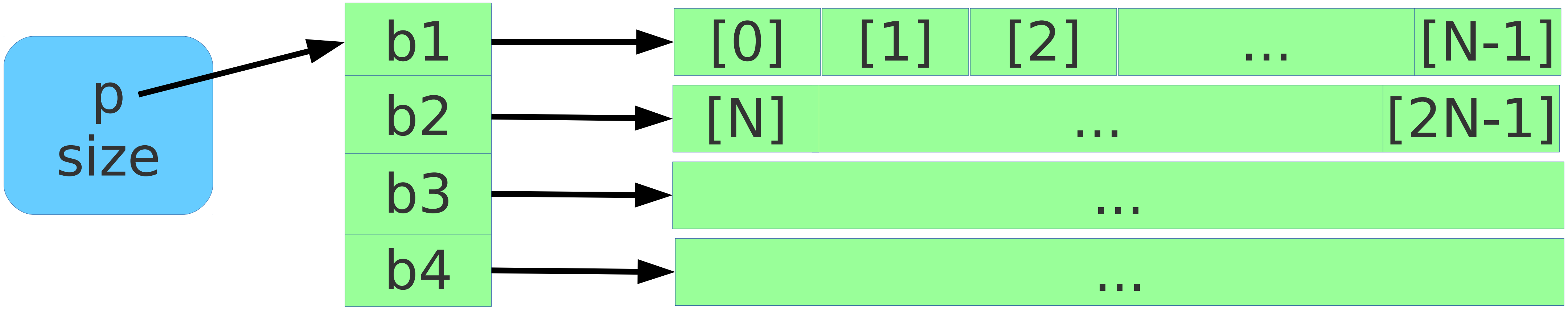
- Solution: block-allocated container (deque)



- Resize adds one or more blocks as needed
- Old blocks never move
- Operator[] needs one indirection, C[i]:
  - read data block pointer from reference block p[i%N]
  - access array element in the data block bx[i/N]

$N=2^m$
No % or /
Use & and >>

Now, with threads

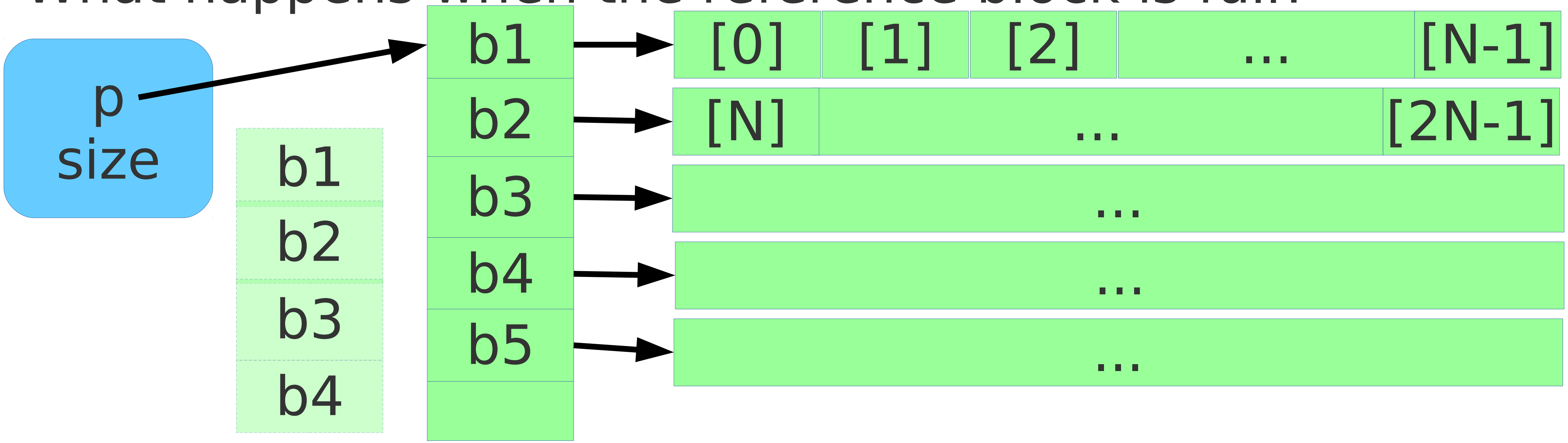


- Writer **publishes** new blocks then updates size (release)
- Reader acquires size, guaranteed access to C[0]...C[size-1]
- Size is the “root pointer” (not p!)
- Resize is locked (or limited to one thread)
- Operator[] is not locked – no overhead for readers

Mentor® A Siemens Business

Keep growing...

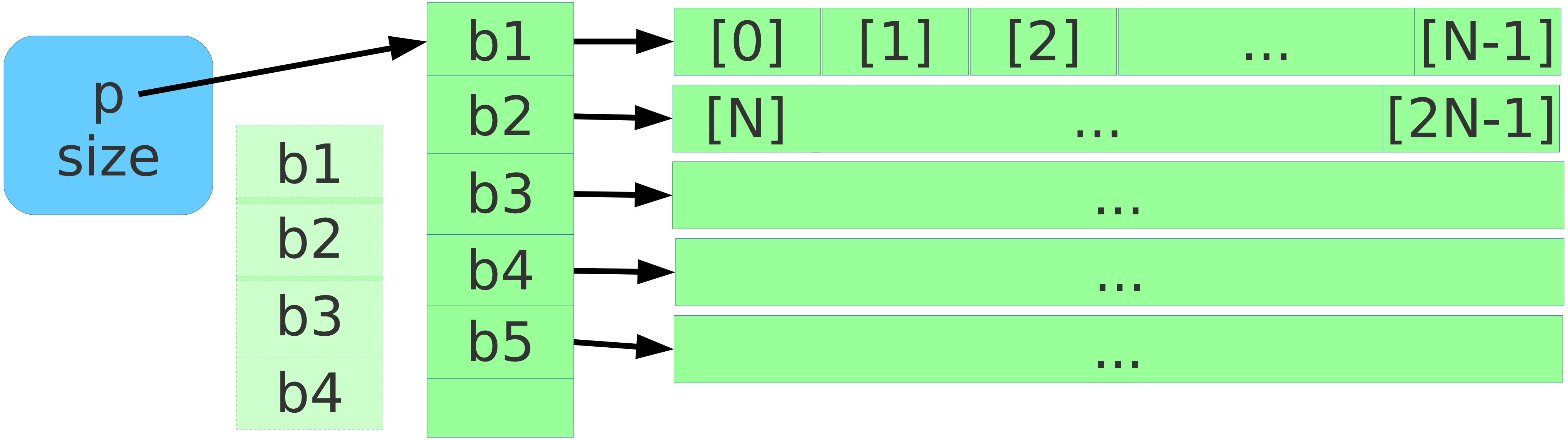
- What happens when the reference block is full?



- Just allocate a larger block, copy the old pointers, add new data block pointer
- RCU – read, copy, update!

RCU in action

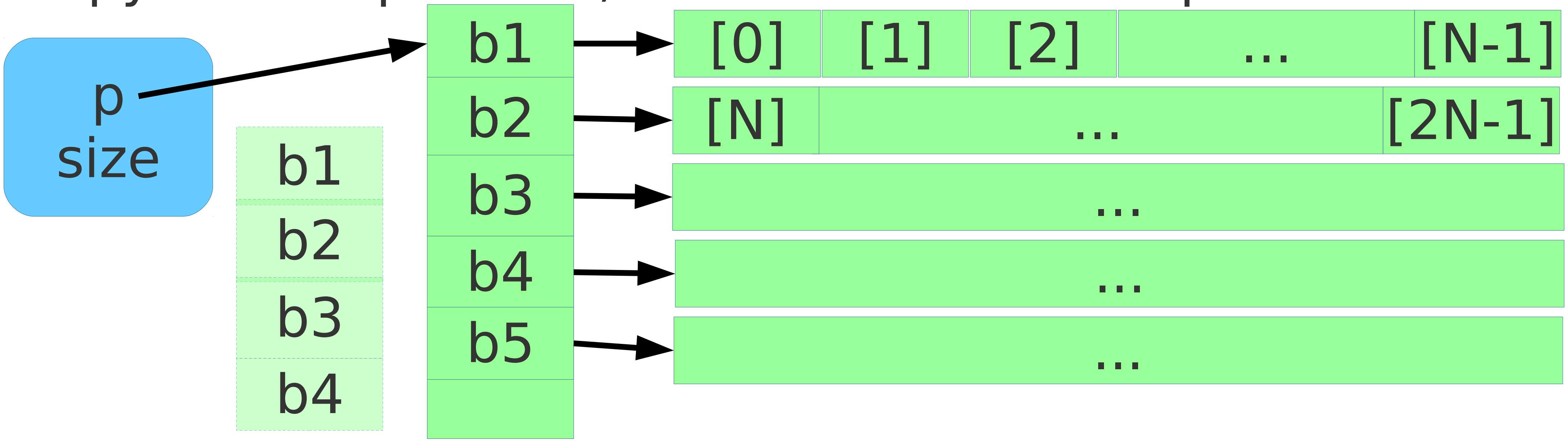
- When the reference block is full, allocate a larger block, copy the old pointers, add new data block pointer
- Still no overhead for readers – RCU rules



- What happens to the old reference block?

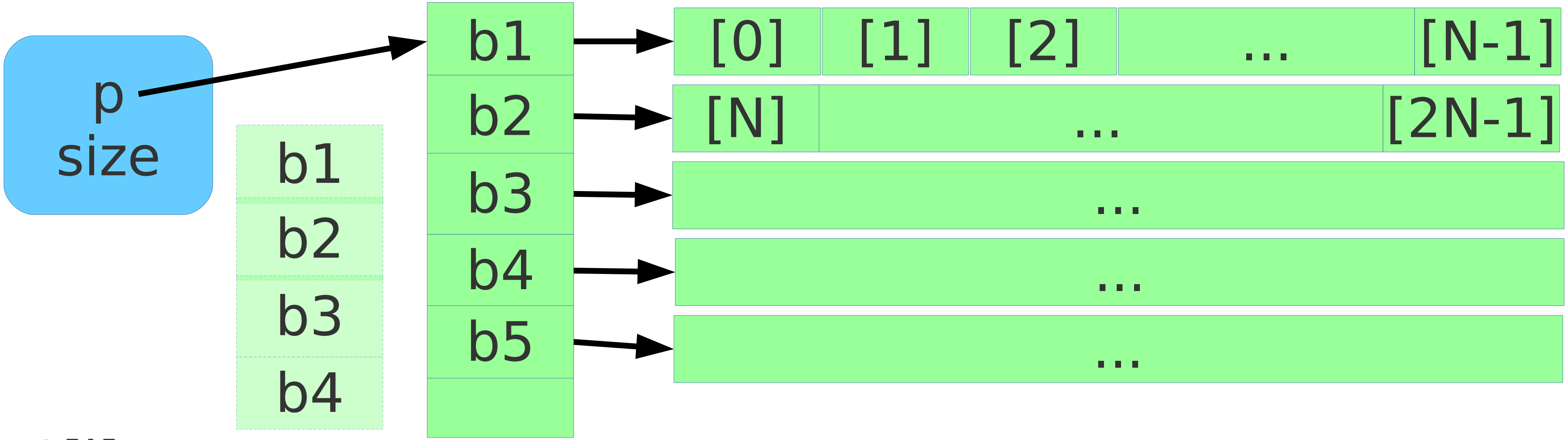
RCU in action

- When the reference block is full, allocate a larger block, copy the old pointers, add new data block pointer



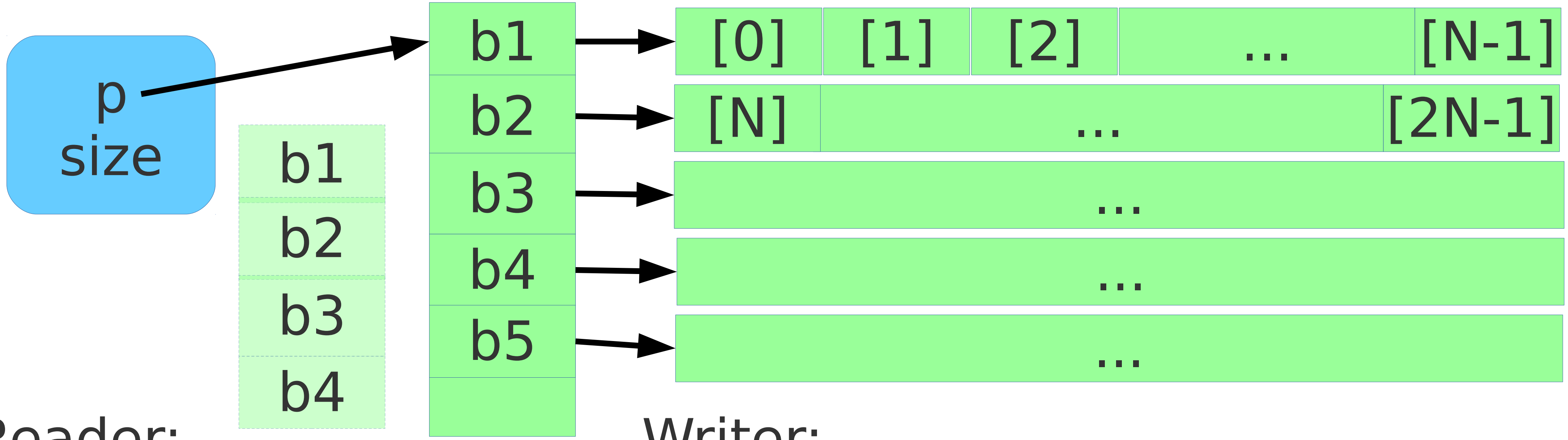
- Old reference block is no longer needed and cannot be reached – delete it

Follow the reader:



- C[i]:
  split i into block index ib=i%N and data index id=i/N
  get bx=p[ib]
  C[i] is bx[id]

Bellevue, we have a problem

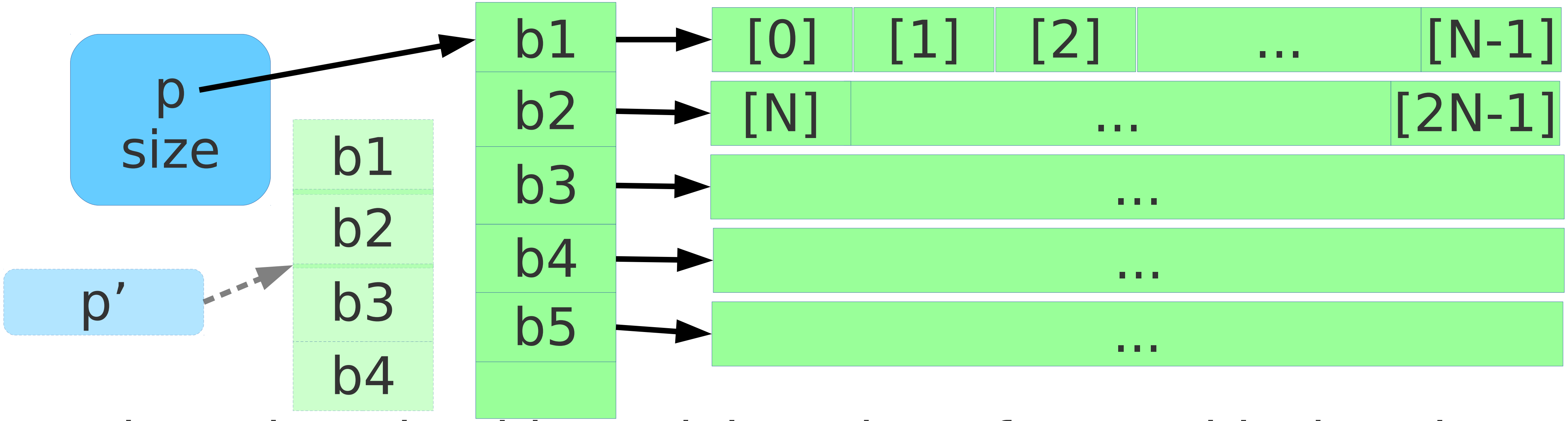


Reader:
compute ib and id
read p (as p')

get bx=p'[ib]
C[i] is bx[id]

Writer:
allocate new reference block
copy old bx pointers to new block
store new p, delete old ref block
seat back and watch the fireworks

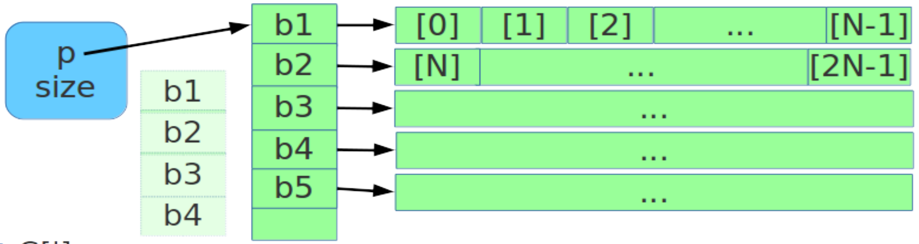
Follow the reader:



- The writer should not delete the reference block as long as at least one reader can access it
- Soon after the resize, all readers will process the indirection p’[ib], then old reference block can be deleted

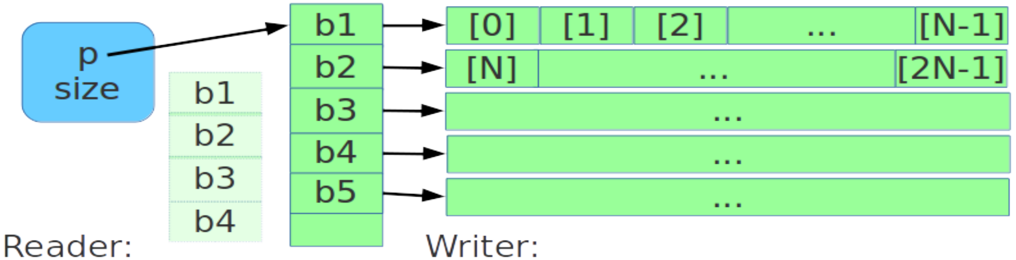
It’s always the delete

Follow the reader:



- C[i]:
  split i into block index ib=i%N and data index id=i/N
  get bx=p[ib]
  C[i] is bx[id]

Bellevue, we have a problem

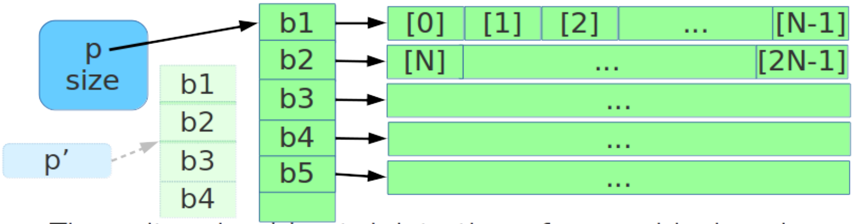


Reader:
compute ib and id
read p (as p’)

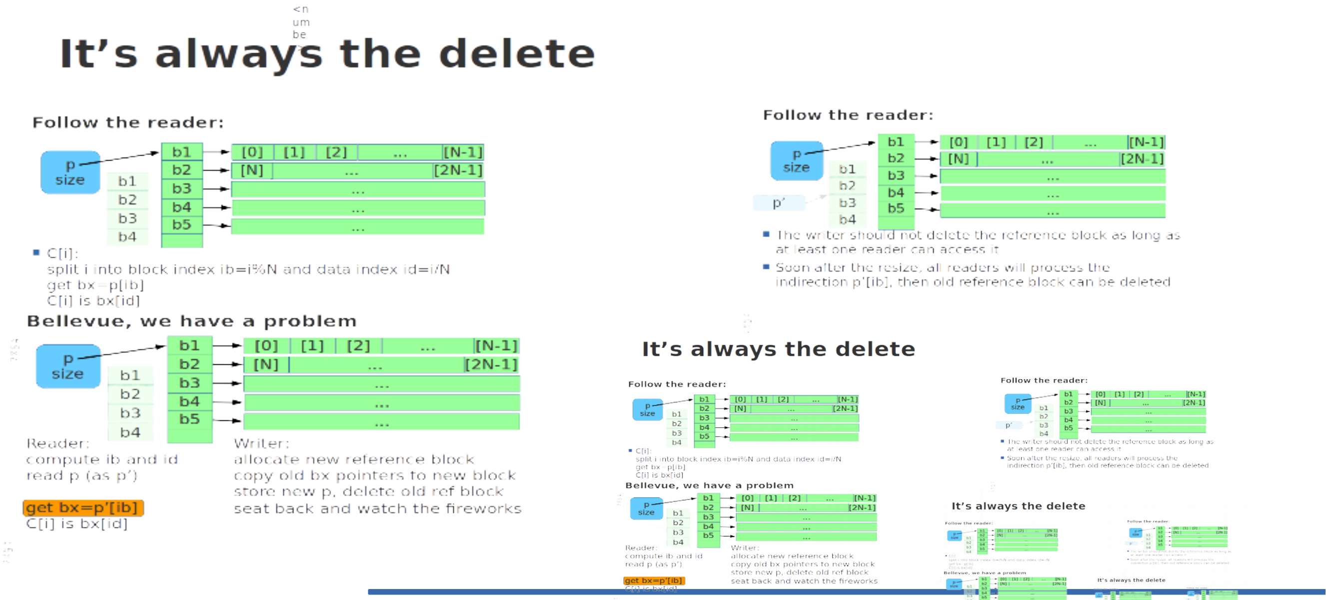
get bx=p’[ib]
C[i] is bx[id]

Writer:
allocate new reference block
copy old bx pointers to new block
store new p, delete old ref block
seat back and watch the fireworks

Follow the reader:



- The writer should not delete the reference block as long as at least one reader can access it
- Soon after the resize, all readers will process the indirection p’[ib], then old reference block can be deleted



It’s always the delete

- Deletion of old data (reclaiming memory) is the main challenge in almost all lock-free programs
  - Inserting data is easy
- The problem is that other threads (readers) may hold “stale” reference to the data that the writer already made inaccessible from the “root pointer”
- The problem is not unique to the RCU but the solution is

Mentor
A Siemens Business

WARNING: confusing terminology ahead!

- "Read, Copy, Update" - the publishing protocol – does not make your concurrency synchronization scheme an RCU
  - You can use publishing protocol with atomic shared pointers or hazard pointers
- RCU is distinguided by the specific way of memory reclamation
- RCU also implies that instead of changing data in place, the writer publishes a new copy of the data
  - Read old data, copy it, and update it
  - Then delete the old data "the RCU way"

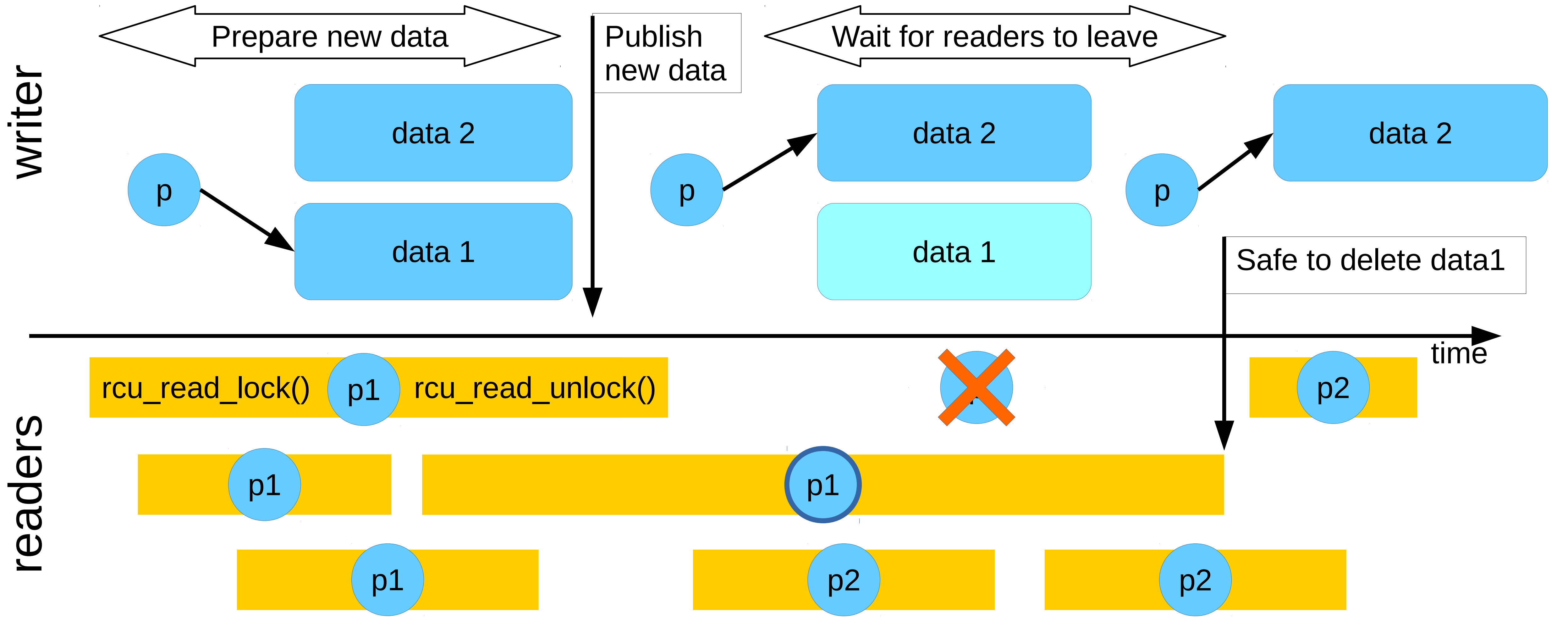
RCU memory reclamation – the real RCU (no Read, Copy, or Update)

- RCU uses cooperative protocol to track when it is safe to reclaim memory (when no reader can access it)
- Readers MUST follow these steps to access shared data:
  1) Call rcu_read_lock() to request access
  2) Get the root pointer (not use the old copy)
  3) Call rcu_read_unlock() to announce end of access
- Readers may access shared data only between the calls to rcu_read_lock() and rcu_read_unlock()
  – reader-side critical section
  – readers in “quiescent state” don’t read shared data

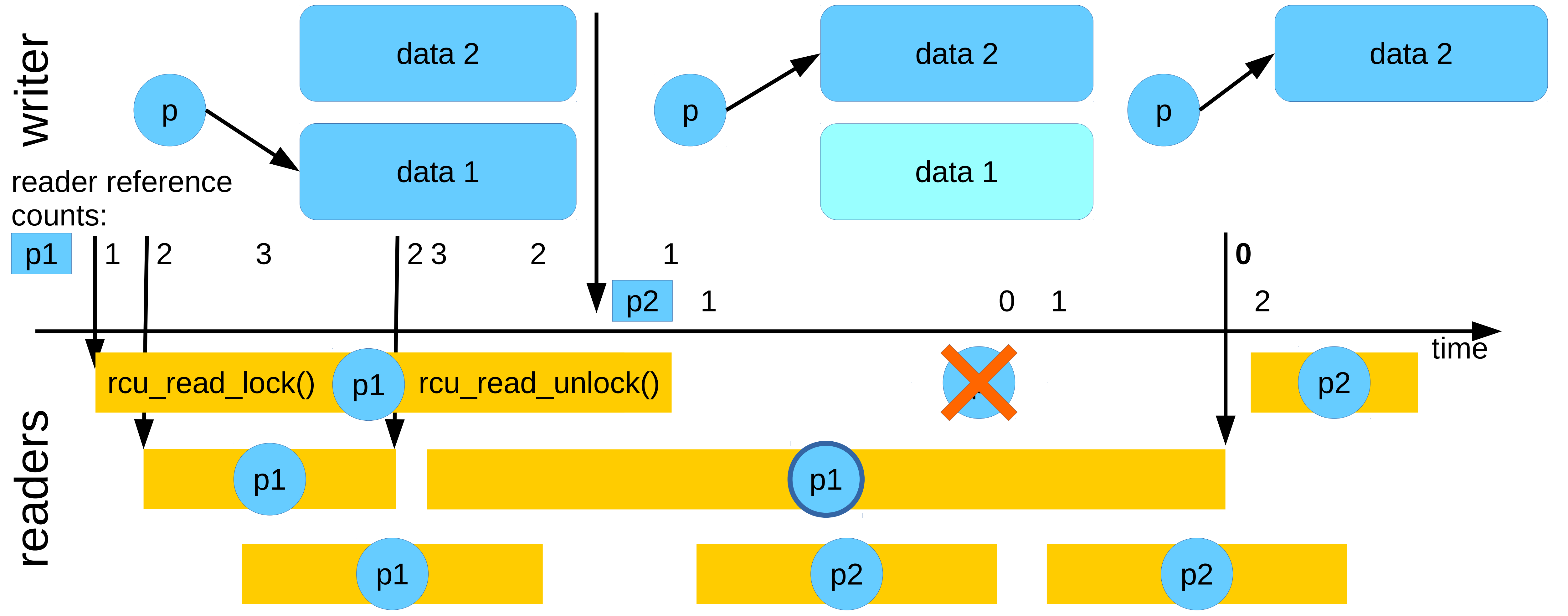
RCU memory reclamation – the real RCU (no Read, Copy, or Update)

- RCU uses cooperative protocol to track when it is safe to reclaim memory (when no reader can access it)
- Writer MUST follow these steps to modify shared data:
  1) Make old shared data inaccessible from the root
  2) Call synchronize_rcu() to wait for all readers who called rcu_read_lock() before step 1 to call rcu_read_unlock()
  3) Delete old data and reclaim the memory
- We don’t need to wait for all readers to exit critical section, only those who acquired the old root pointer

RCU protocol



RCU implementation – the main idea



RCU under any other name

- RCU implementations use different function names:
- Readers entering critical section: rcu_read_lock, rcu_enter
- Readers leaving critical section: rcu_read_unlock, rcu_leave
- Writer waiting for readers before reclaiming memory: synchronize_rcu, wait_for_readers_to_leave
- C++ standard proposal WG21/P0461R1 uses names rcu_read_lock, rcu_read_unlock, and synchronize_rcu
- RCU implementations may use additional functions to register threads, defer deletion callbacks, etc

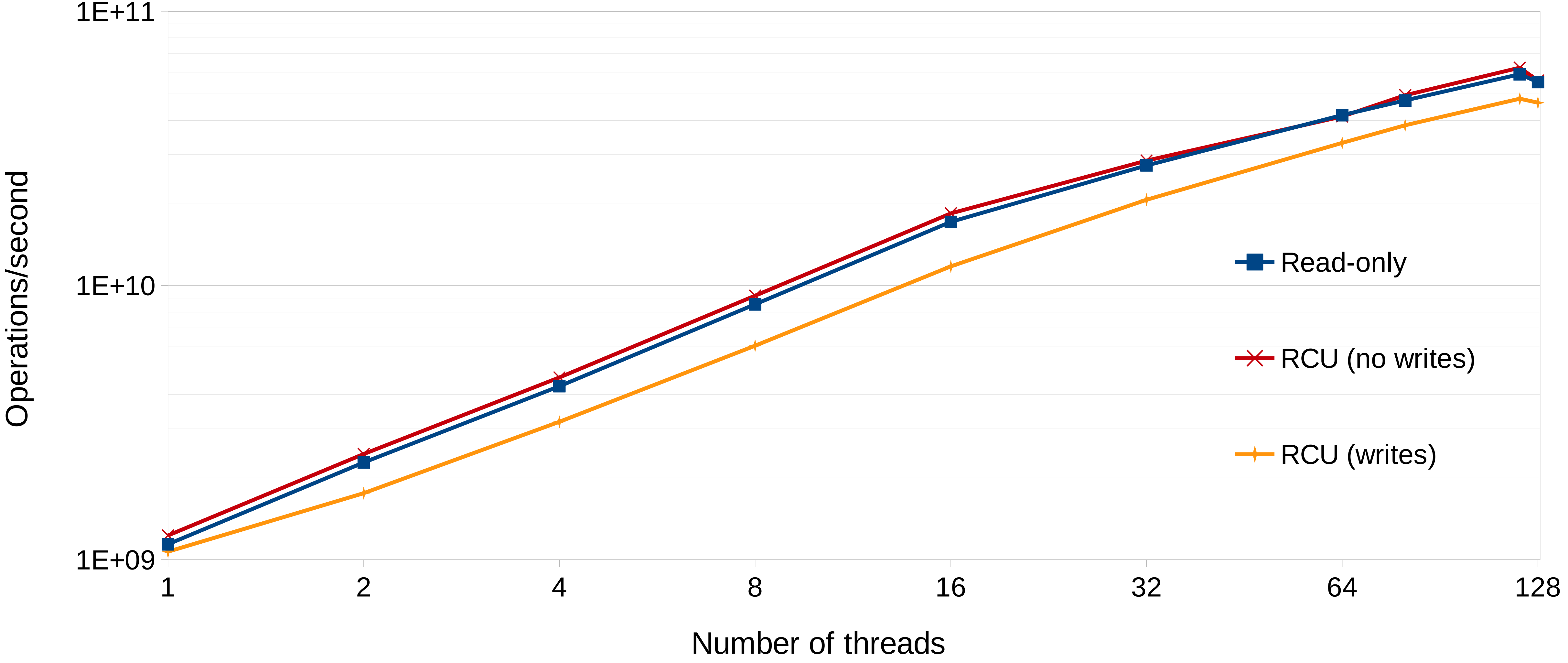
User-space RCU vs kernel RCU

- RCU is used in Linux kernel extensively
- Kernel RCU has several advantages
  - Kernel knows when context switches happen (readers leave critical section)
  - Kernel knows how many threads are there, which ones are running, etc
  - Kernel RCU does not need (extra) memory barriers
- The basic idea is the same but it’s important to understand the implicit assumptions

RCU vs GC

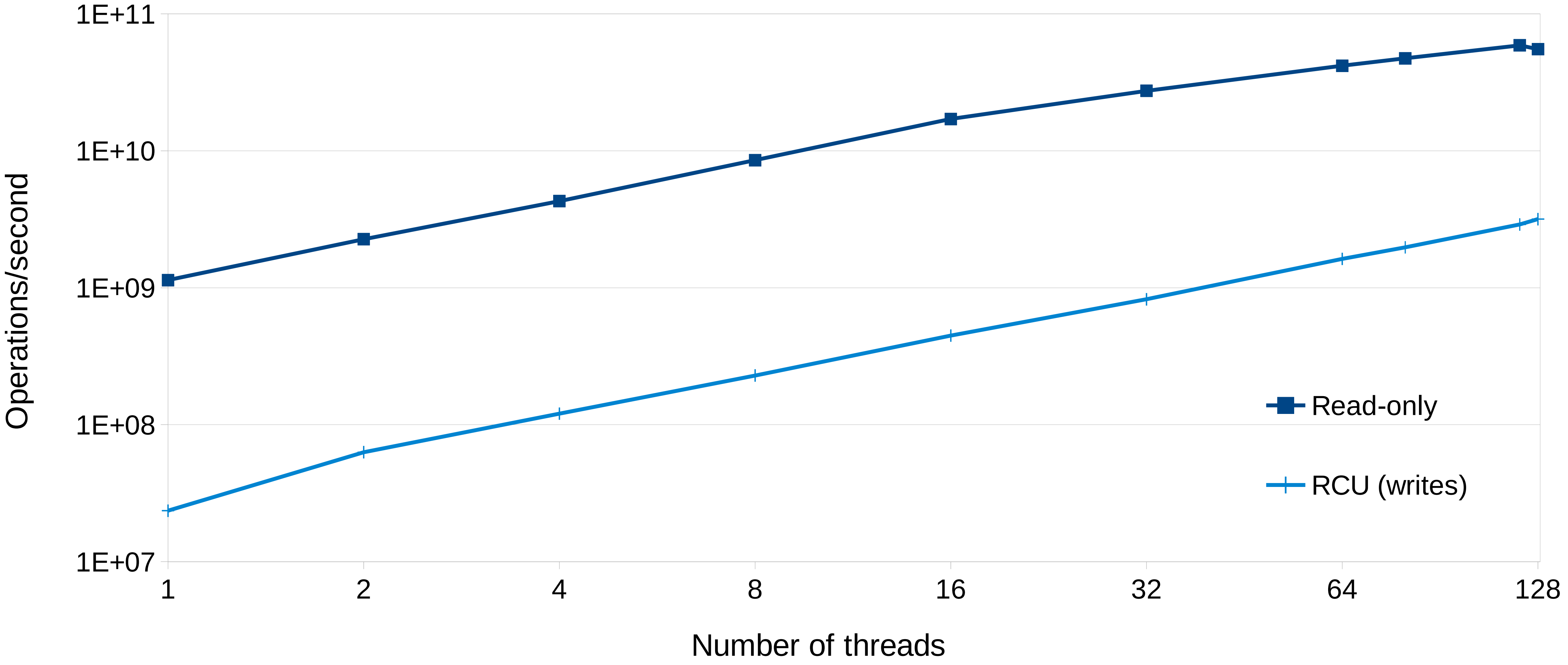
- RCU does memory reclamation when that memory is unreachable (just like garbage collection)
- Strictly speaking, GC implies “automatic” (by definition)
  - “User-driven GC” is a contradiction in terms
- “User-driven GC” is a term that’s often used and I don’t know of a good alternative term
  - E.g. allocate a lot of objects on a memory pool then free the entire pool without deallocating each object
  - Memory reclamation in RCU is a kind of user-driven GC

Searching read-only (or not) data with RCU



Mentor
A Siemens Business

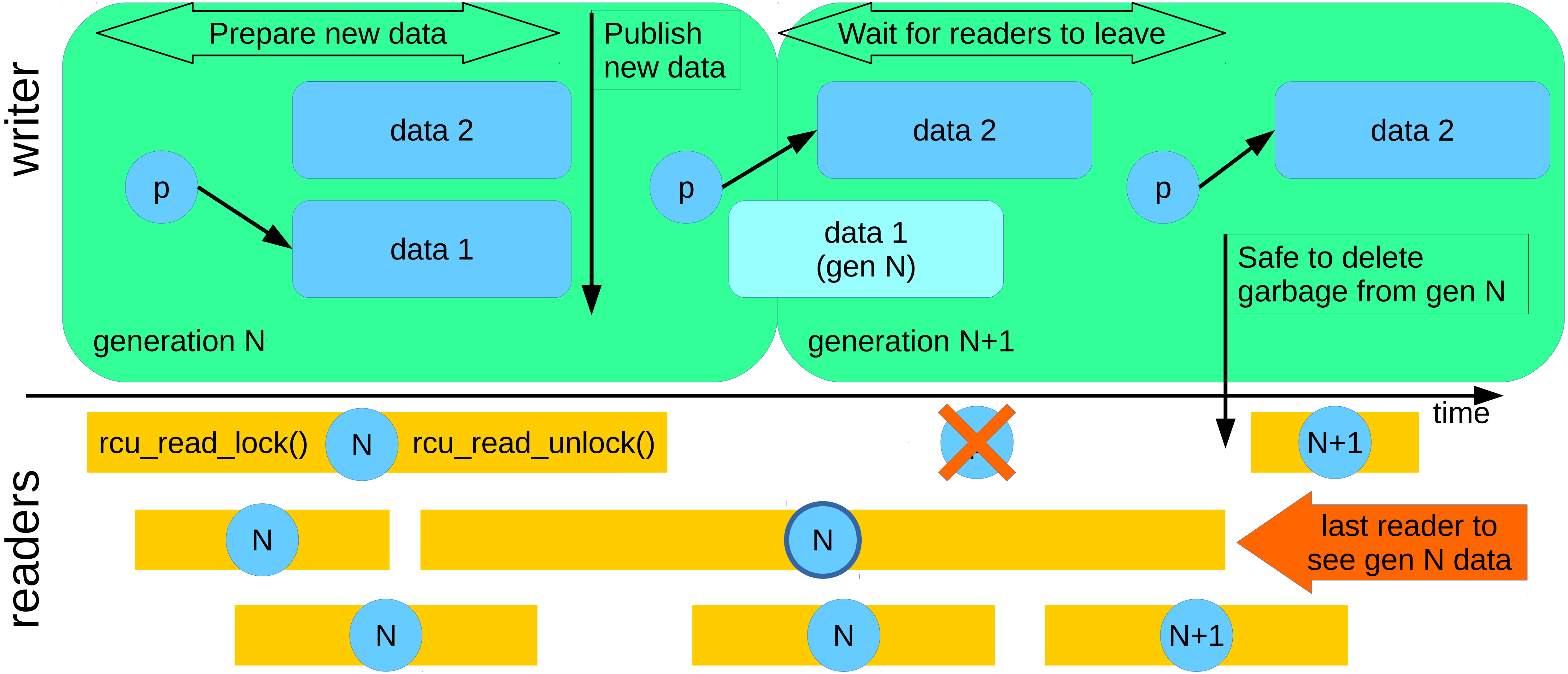
Searching data with (another) RCU



User-space RCU implementation (one of many)

- Writer maintains a global atomic generation (epoch)
  std::**atomic**<unsigned long> **generation**;
- All data currently live belong to the current generation
- When the writer does an update, the old data is placed in a garbage queue
- There is one garbage queue per generation
- At some point the writer increments the generation
- Readers can access only current generation data
  - But can keep accessing it after it becomes garbage
- Writer tracks how many readers access each generation

User-space generational RCU



User-space RCU implementation: Reader interface

```
std::atomic<unsigned long> generation;
std::atomic<unsigned long> refcount[max_generations];
class handle_t {…}; // Contains reader generation
handle_t rcu_read_lock() {
  size_t cg=generation;
  ++refcount[cg];
  return handle_t(cg);
}
void rcu_read_unlock(handle_t handle) {
  --refcount[handle.get()];
}
```

User-space RCU careful implementation: Reader interface

```
handle_t rcu_read_lock() {
  size_t cg=generation;
  ++refcount[cg];
  return handle_t(cg);
}
```

- But aren’t atomics memory_order_seq_cst by default? And isn’t memory_order_seq_cst expensive? - Yes, and can be.
- Real implementation will use acquire/release fences and memory_order_relaxed as much as possible
  - Minimum necessary barriers depend on details of the implementation (one sec_cst is usually unavoidable)

User-space RCU implementation: Writer interface

```
std::atomic<unsigned long> generation;
std::atomic<unsigned long> refcount[max_generations];
std::queue<data_t*> garbage[max_generations];
unsigned long last_gc_gen = 0;
void synchronize_rcu() {
  unsigned long last_gen = generation++;
  while (last_gc_gen < last_gen) {
    while (refcount[last_gc_gen] > 0) wait();
    delete_garbage(garbage[last_gc_gen]);
    ++last_gc_gen;
  }
}
```


Mentor
A Siemens Business

RCU implementations

- All RCU implementations will have rcu_read_lock(), rcu_read_unlock(), and synchronize_rcu()
  - Or enter(), leave(), and wait_for_readers_to_leave()
- Then you can get creative...

RCU implementations: readers

- Reader implementations may try to minimize read_lock() and read_unlock() overhead
  - Possibly by imposing some restrictions
- Why reader overhead?
  - Read generation number
  - Increment reference count ← read-modify-write shared data
  - Read root pointer ← acquire memory barrier
- Cache contention on the reference count is the main overhead (in user-space RCU)

RCU implementations: readers

- Reference count is shared between all readers
- Possible solution: give each reader its own count

```
handle_t rcu_read_lock(size_t reader_id) {
  size_t cg=generation;
  ++refcount[reader_id][cg]; // Padded to cache line
  return handle_t(cg);
}
```

- Reader ID is a thread ID: 0, 1, 2…
- Reader threads must register in advance with the RCU
- Writer must loop over all reader slots to add up the count

RCU implementations: readers

- Can reader critical sections be nested? If not, each reader may access only one version of shared data at a time, no need for a reference count array:

```
void rcu_read_lock(size_t reader_id) {
  size_t cg=generation;
  reader_gen[reader_id] = cg; // Padded to cache line
}
const size_t NO_READER = 0; // ULONG_MAX, etc
void rcu_read_unlock(size_t reader_id) {
  reader_gen[reader_id] = NO_READER;
}
```

RCU implementations: readers

- What if thread registration is impossible?
- Can still reduce reader contention by using an array of reference counts, index is a hash of thread ID
- Hash collisions can happen, so reference counts must be atomically incremented
  - But contention is greatly reduced
  - Remember false sharing, one count per cache line!

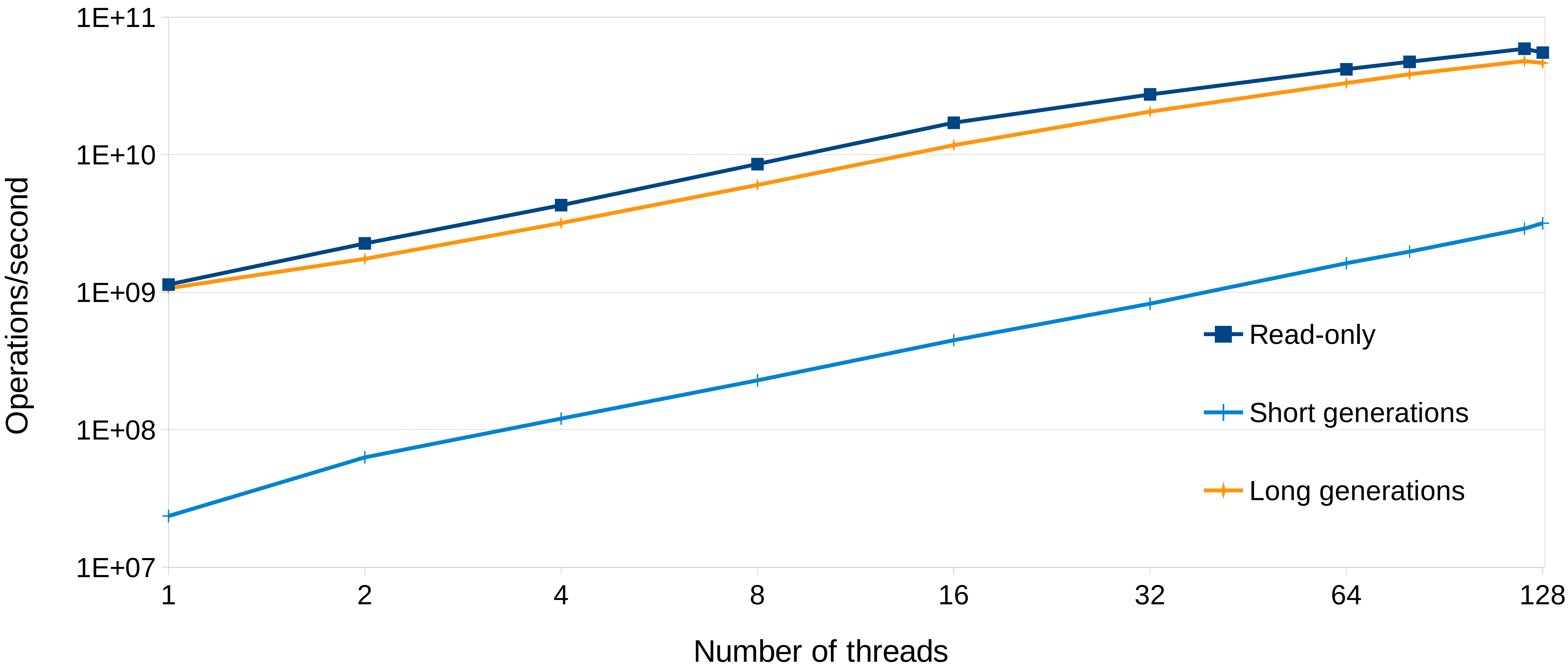
RCU implementations: writers

- synchronize_rcu() vs call_rcu()
- RCU writer may defer deletion callbacks and queue them:
  data_t* old_data = current_data;
  data_t* new_data = new data_t(*old_data);
  new_data->update();
  current_data = new_data;
  call_rcu(old_data, deleter); // Deferred until readers leave
- Callbacks are executed in batches by the writer thread or a special cleanup thread
  - Deletion callbacks are deferred until all readers leave the generation to which callbacks belong

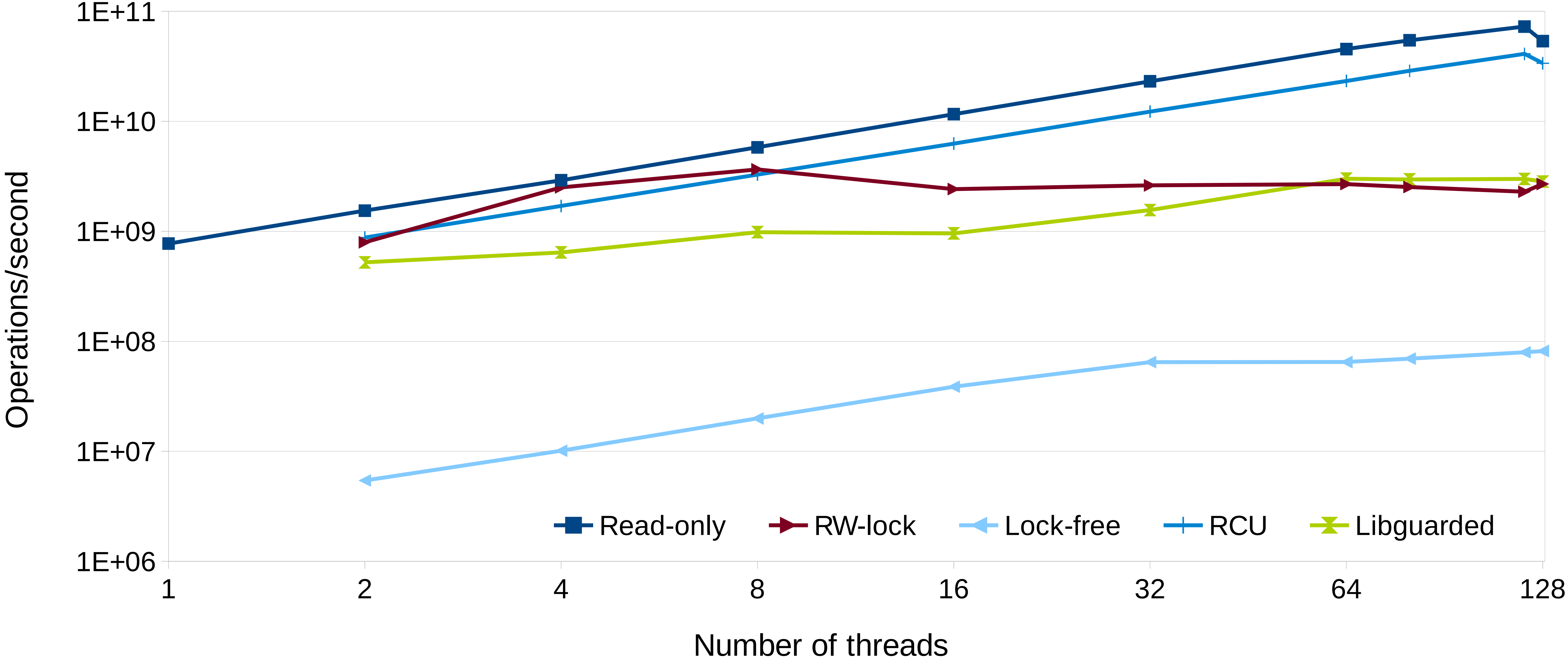
RCU implementations: writers

- Granularity of updates vs granularity of cleanup
- Writer may process many updates in one generation
  - Old data from earlier updates cannot be reclaimed until generation changes
  - readers can enter current generation any time, only old generations are not re-readable once all readers leave
- Guard object – each update of each root pointer effectively advances the generation
  - More overhead, less unclaimed garbage

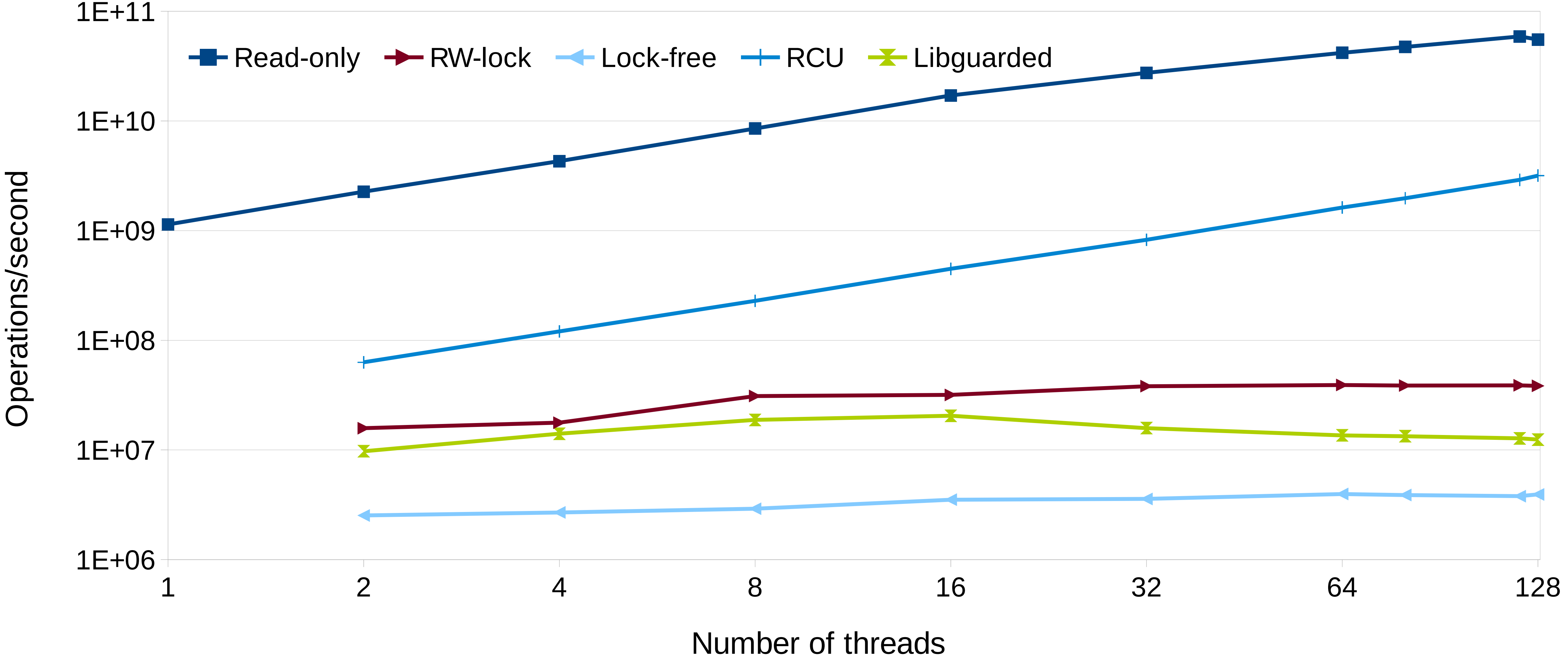
Short-lived vs long-lived generations



RCU performance – slowly changing data



RCU performance – rapidly changing data



Mentor
A Siemens Business

RCU implementations: writers

- Readers do not wait for writers, ever – check
- Writer can do updates without waiting for readers – check
- Eventually writer will run out of memory…
- Ideally writer can reclaim memory between updates
- No-wait synchronize_rcu(), or GC(), or do_callbacks() - reclaim memory that no reader can see, right now
  - If the garbage queue is long, process a batch then return, to avoid delaying updates
- wait_for_readers_to_leave(timeout) – blocking wait with a timeout, reclaim what you can while waiting

RCU implementations: writers

- RCU itself does nothing for multiple writers
  - RCU works well when updates are infrequent – one writer thread should be able to handle the load
  - Or one writer thread at a time – lock or CAS
- Preparing new data may be time-consuming, so multiple writer threads are often desirable
- Some implementations support concurrent calls to synchronize_rcu() - makes multiple writers easier
  - Garbage queue may be per-thread or global, per-generation or sentinel-delimited

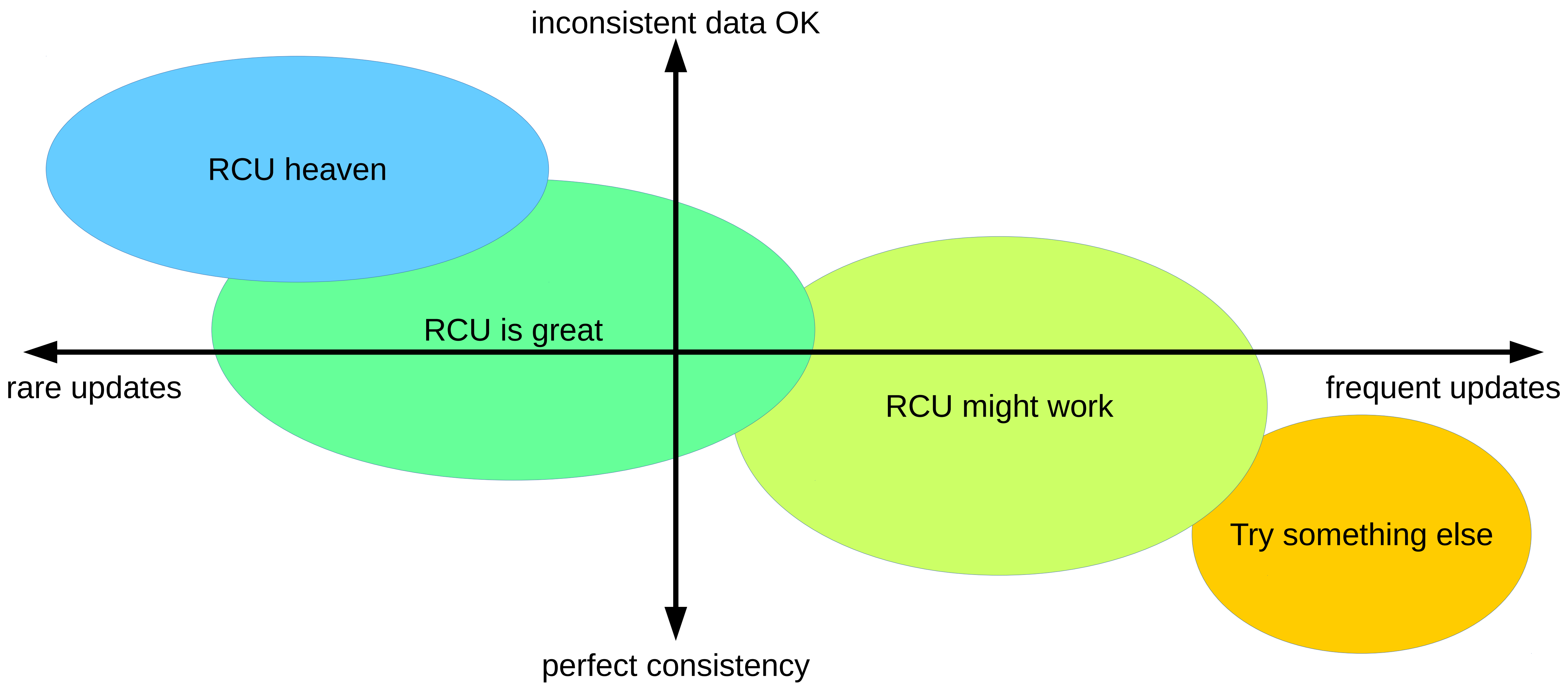
RCU implementations: writers and readers

- Reader can hold a reference to the shared data forever
  - Writer can avoid blocking waits but cannot reclaim data
- In some applications writers can force-reclaim (after a grace period) – readers will be left in an undefined state

RCU implementations: mean writers and meek (or just fault-tolerant) readers

- Memory is reused but not freed, readers can handle inconsistent data – RCU paradise
- Memory is reused, readers need consistent data
  bool rcu_read_unlock(handle_t handle);
  - Returns false if the handle has expired
  - Everything done in this critical section must be redone
- Readers can crash and it’s OK
  - Separate reader processes, stateless, easy to restart

When to use RCU



RCU and alternatives

	RCU	Hazard pointers	Atomic shared_ptr
Readers	wait-free population-oblivious	lock-free (wait-free?)	lock-free (slow)
Writers	single writer (BYOL)	lock-free	lock-free
Reclamation	blocking (or memory grows)	non-blocking	lock-free
Garbage	unbounded (or writers must block)	bounded by $Nthreads*N_{HP}$	None
Ease of use	very easy	hard	easy (watch out for cycles)

```
rcu_questions_unlock();
do { answer_questions() }
while ( !wait_for_readers_to_leave(); )
```

Mentor®

A Siemens Business

www.mentor.com