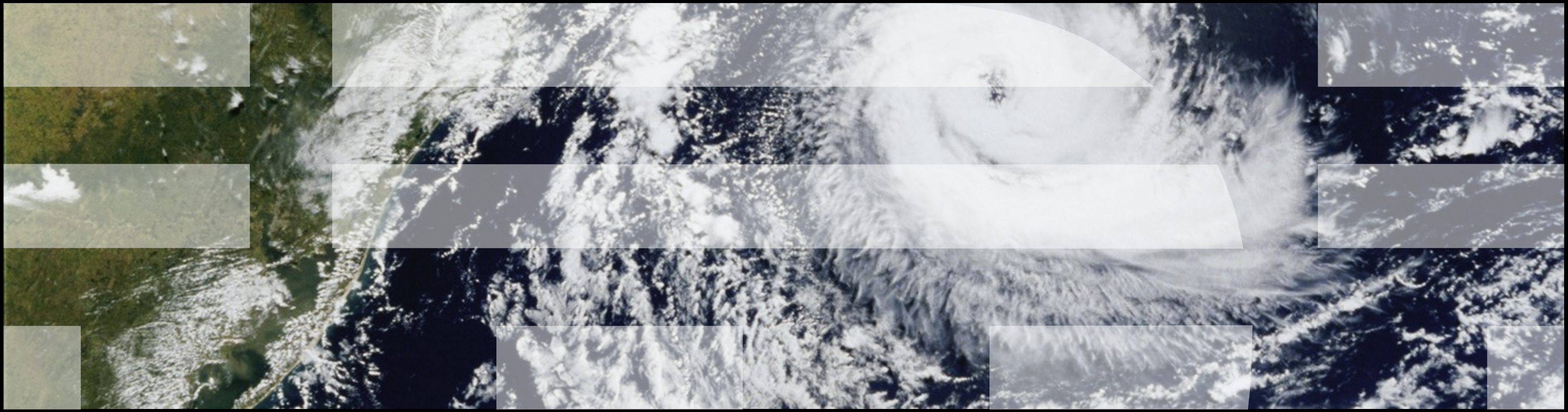
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center

Member, IBM Academy of Technology

CMU CS 410 Operating Systems, September 20, 2017

What Is RCU?



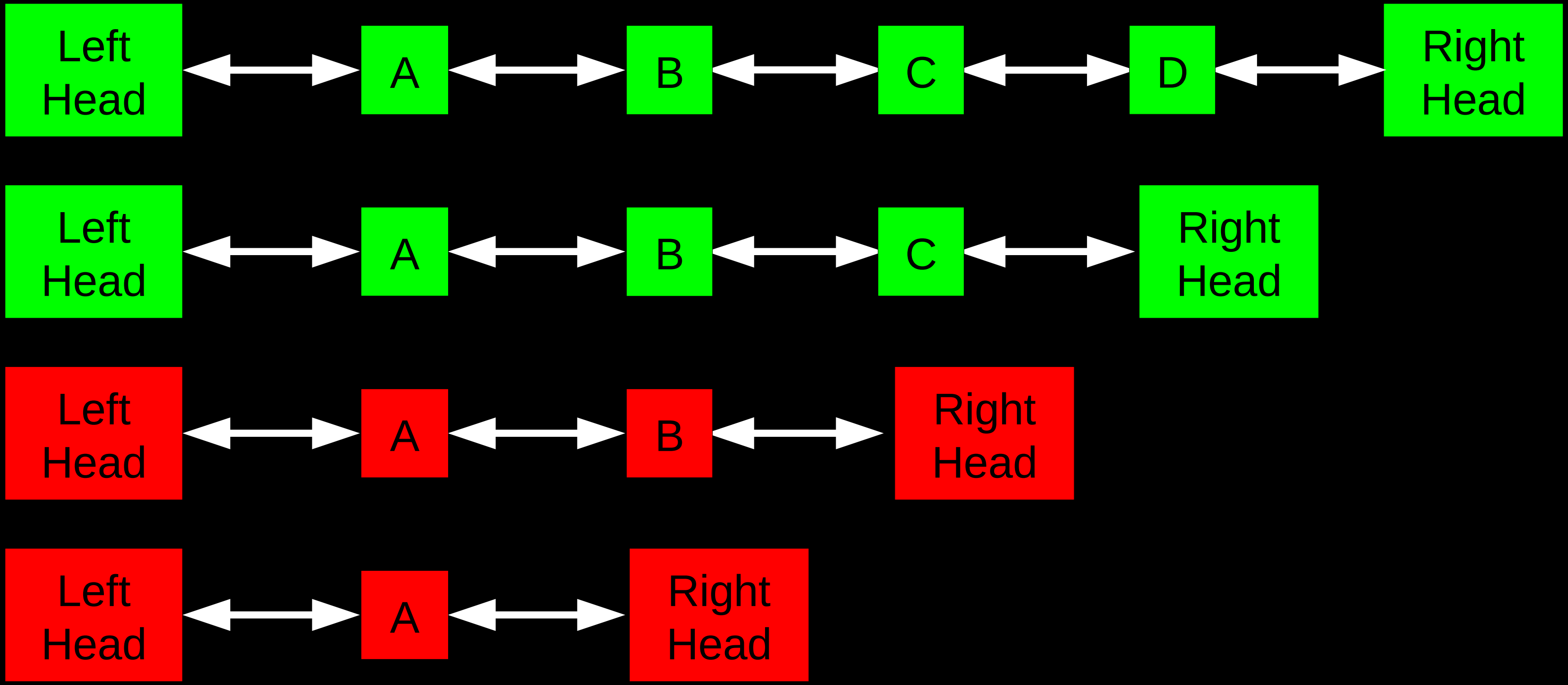
Overview

- Mutual Exclusion
- Example Application
- Performance of Synchronization Mechanisms
- Making Software Live With Current (and Future) Hardware
- Implementing RCU
- RCU Grace Periods: Conceptual and Graphical Views
- Performance
- RCU Area of Applicability
- The Issaquah Challenge
- Summary

Mutual Exclusion

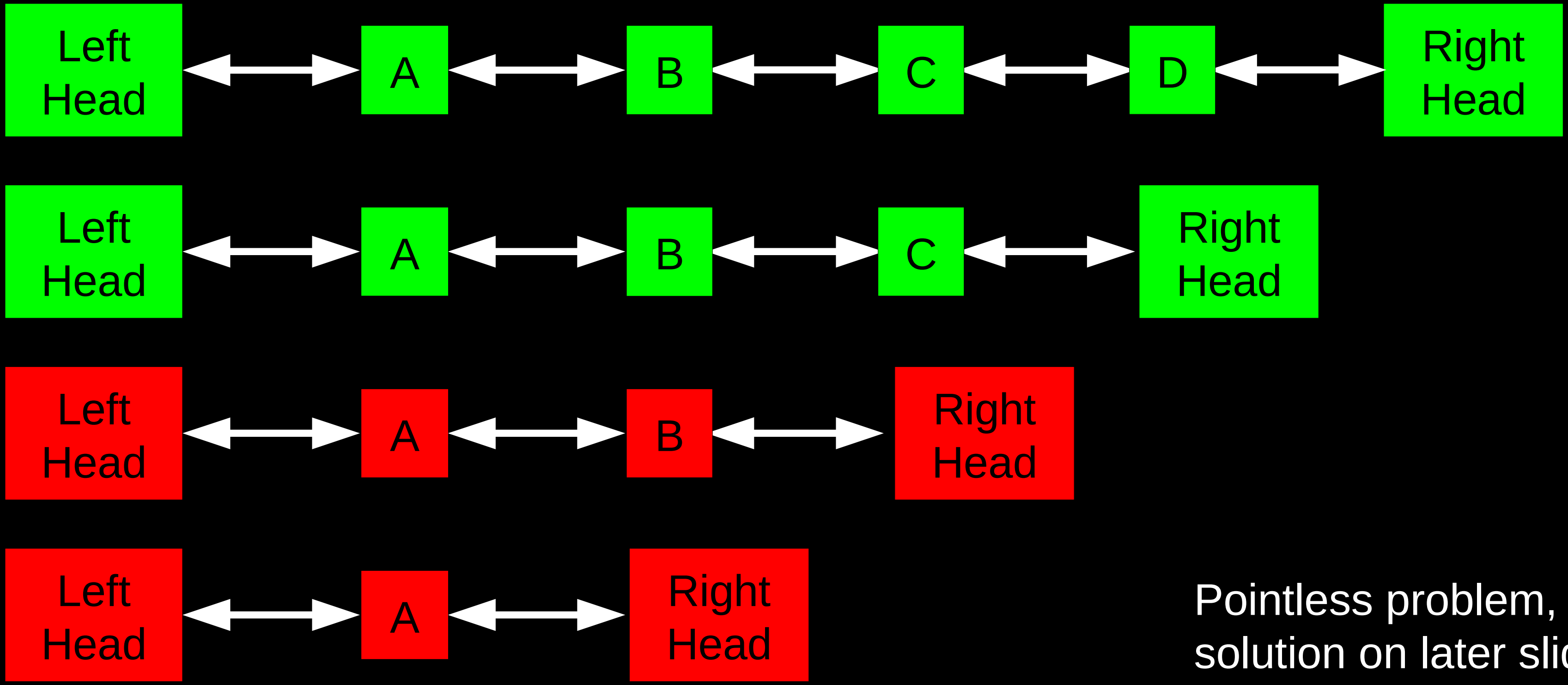
Mutual Exclusion Challenge: Double-Ended Queue

- Can you create a trivial lock-based deque allowing concurrent pushes and pops at both ends?
  - Coordination required if the deque contains only one or two elements
  - But coordination is not required for three or more elements



Mutual Exclusion Challenge: Double-Ended Queue

- Can you create a trivial lock-based deque allowing concurrent pushes and pops at both ends?
  - Coordination required if the deque contains only one or two elements
  - But coordination is not required for three or more elements



Pointless problem, but solution on later slide...

Mutual Exclusion Question

- What mechanisms can enforce mutual exclusion?

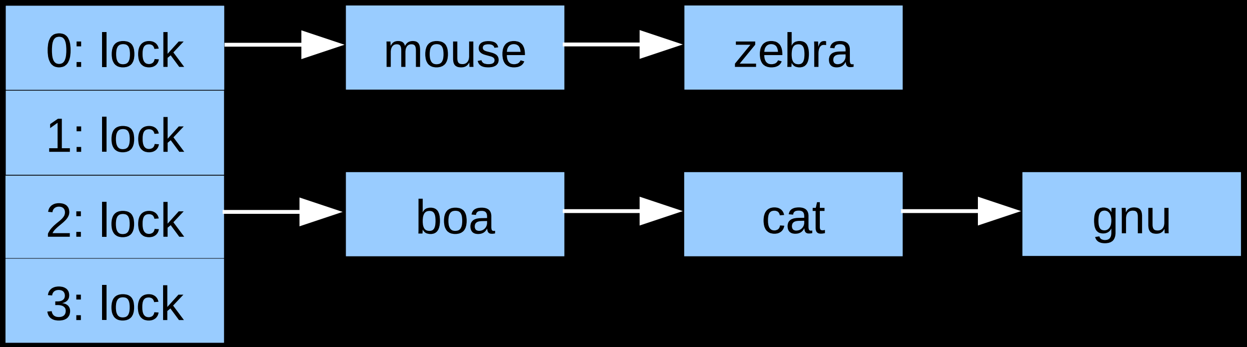
Example Application

Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)

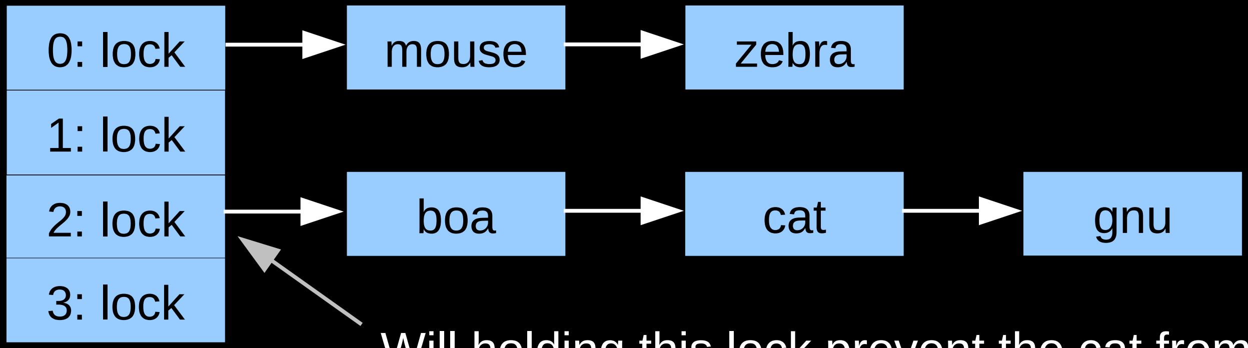
Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)
- Simple approach: chained hash table with per-bucket locking



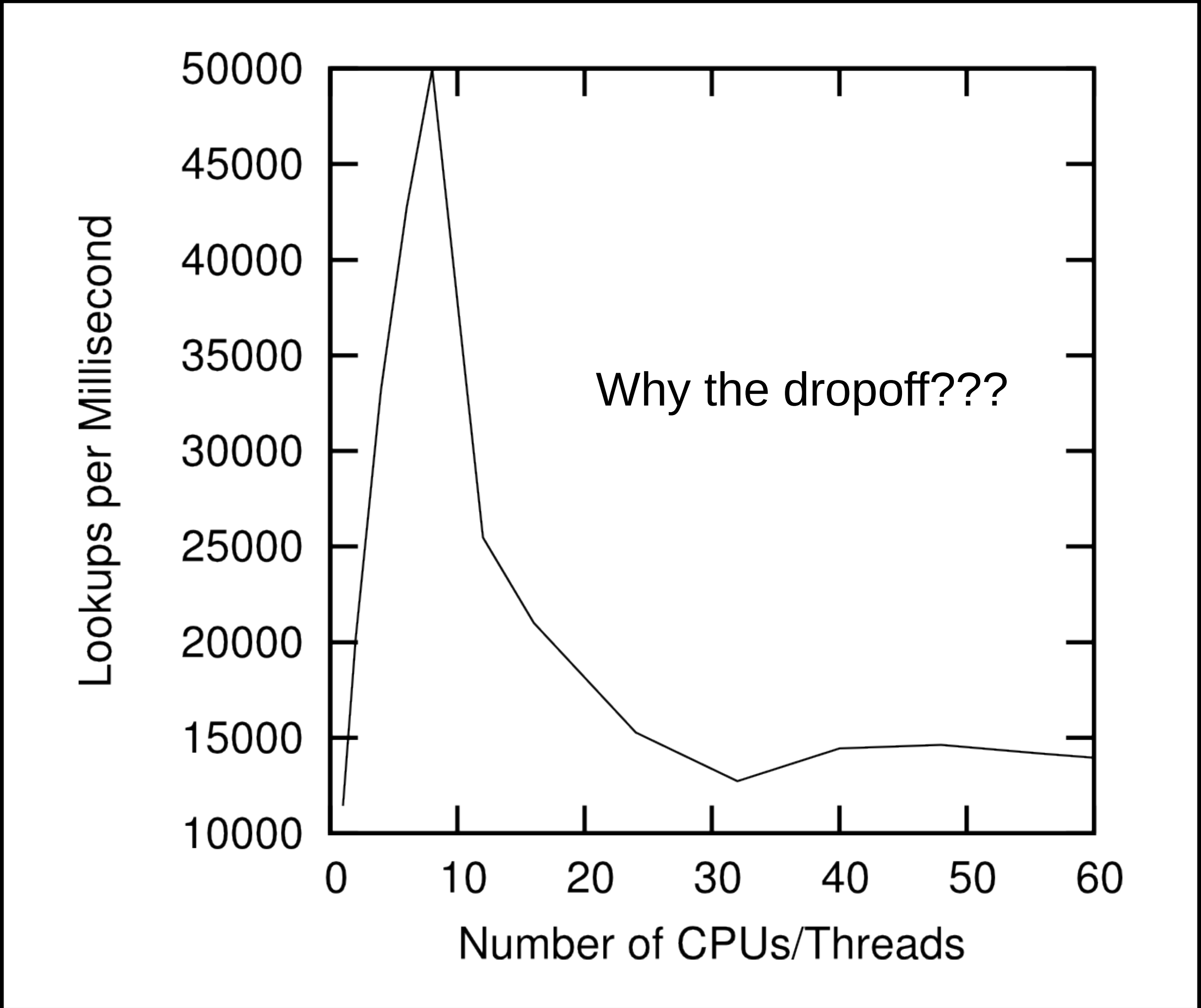
Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)
- Simple approach: chained hash table with per-bucket locking



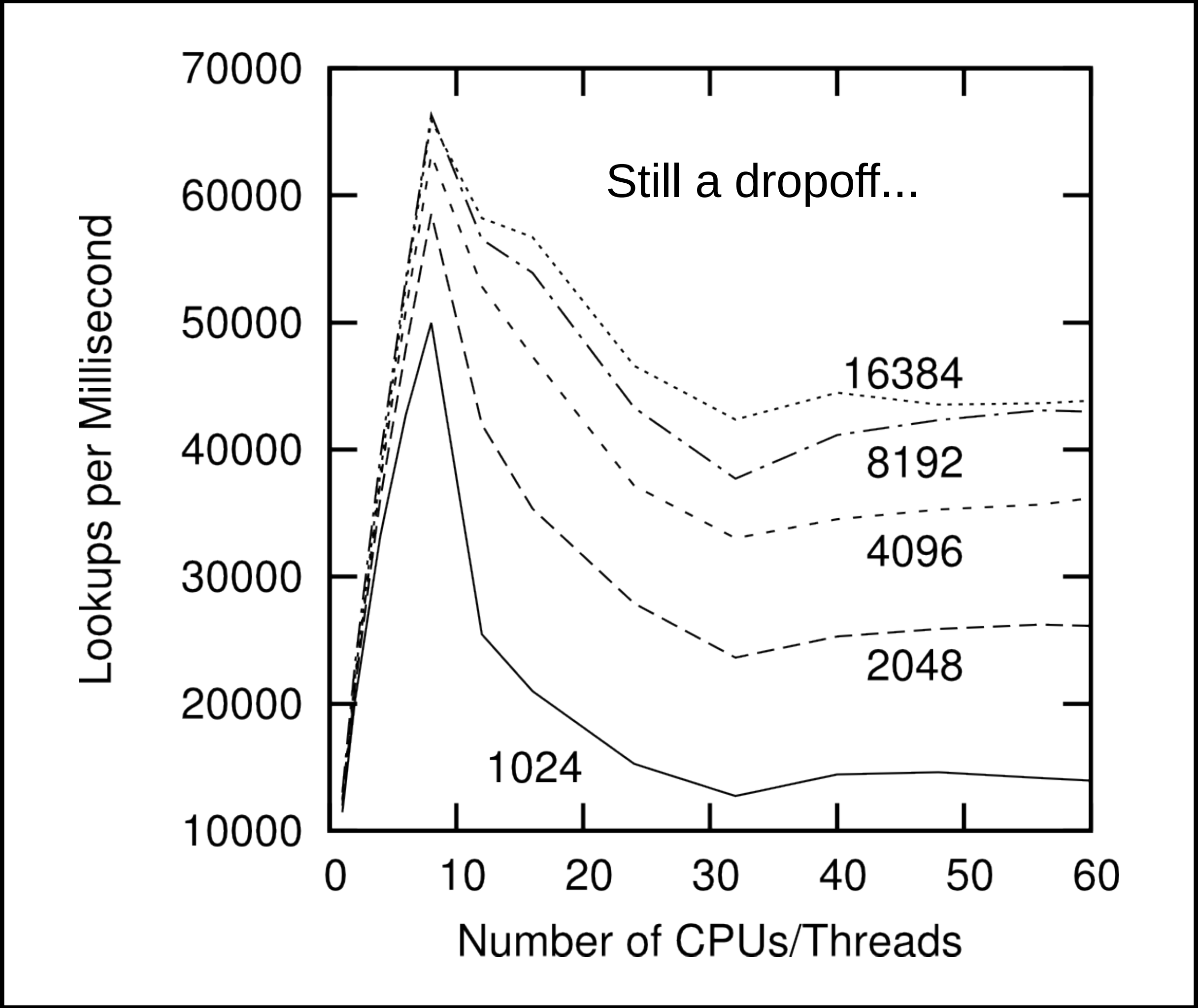
Will holding this lock prevent the cat from dying?

Read-Only Bucket-Locked Hash Table Performance



2GHz Intel Xeon Westmere-EX, 1024 hash buckets

Varying Number of Hash Buckets



Still a dropoff...

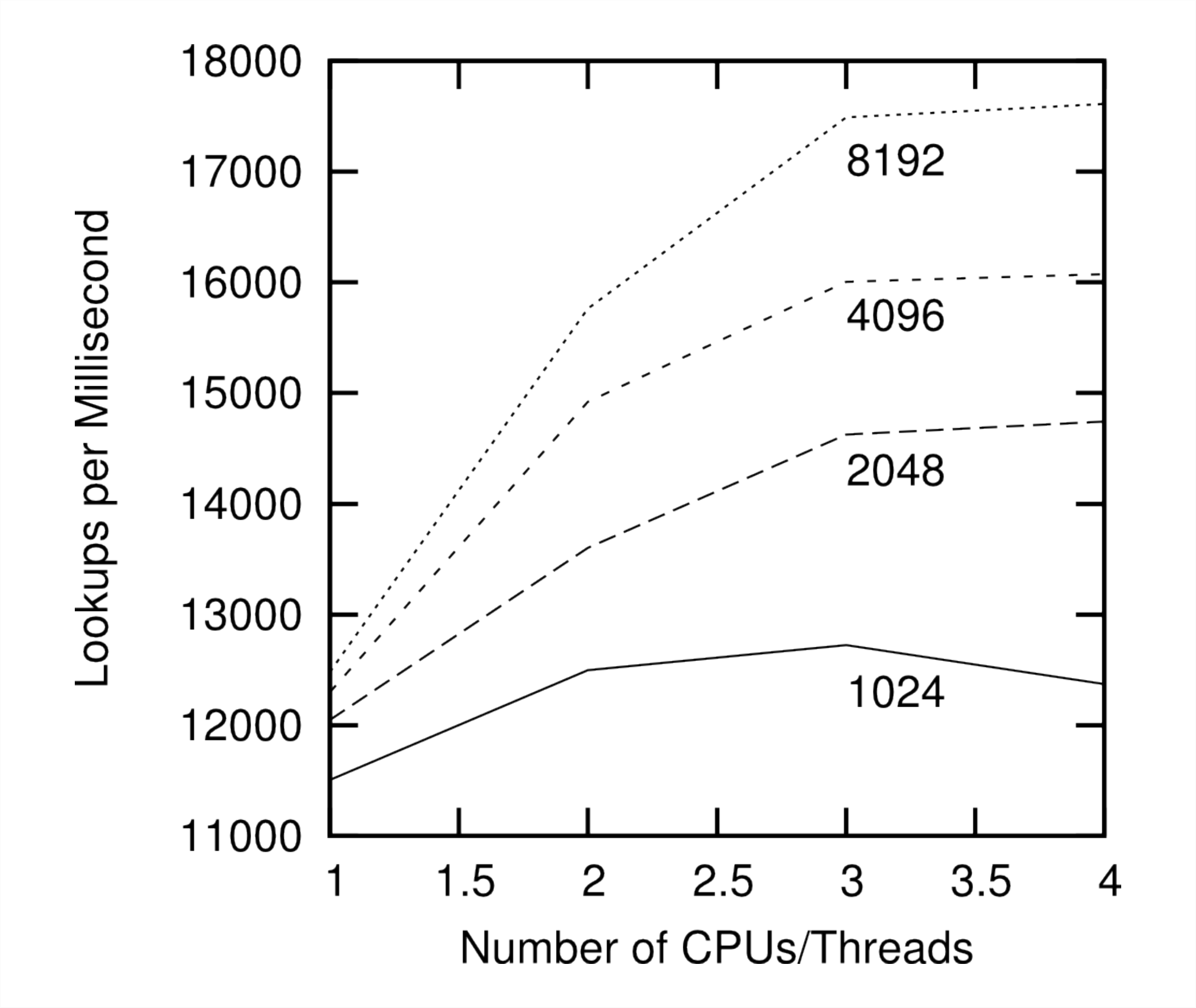
Some improvement, but...

2GHz Intel Xeon Westmere-EX

NUMA Effects???

- /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index1/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list:
  - **0-7,32-39**
- Two hardware threads per core, eight cores per socket
- Try using only one CPU per socket: CPUs 0, 8, 16, and 24

Bucket-Locked Hash Performance: 1 CPU/Socket

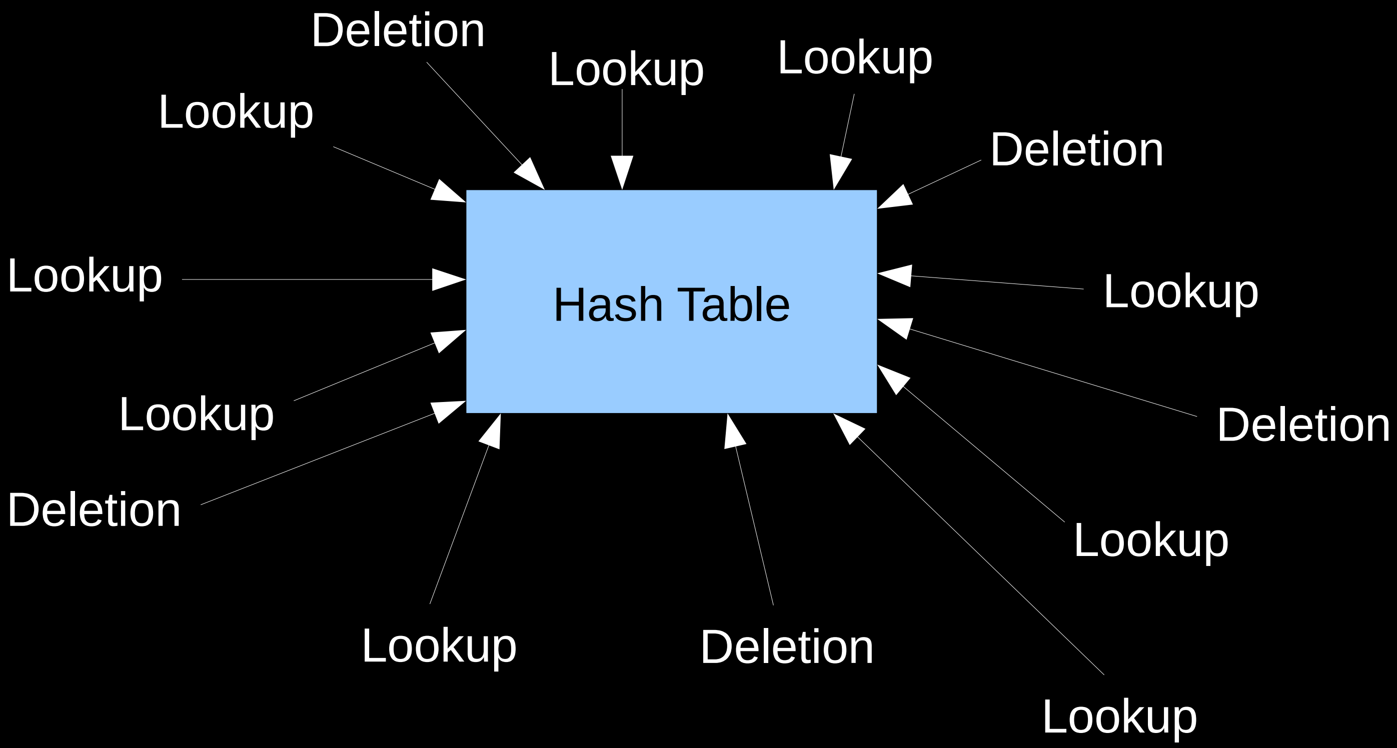


This is not the sort of scalability Schrödinger requires!!!

Locking is BAD: Use Non-Blocking Synchronization!

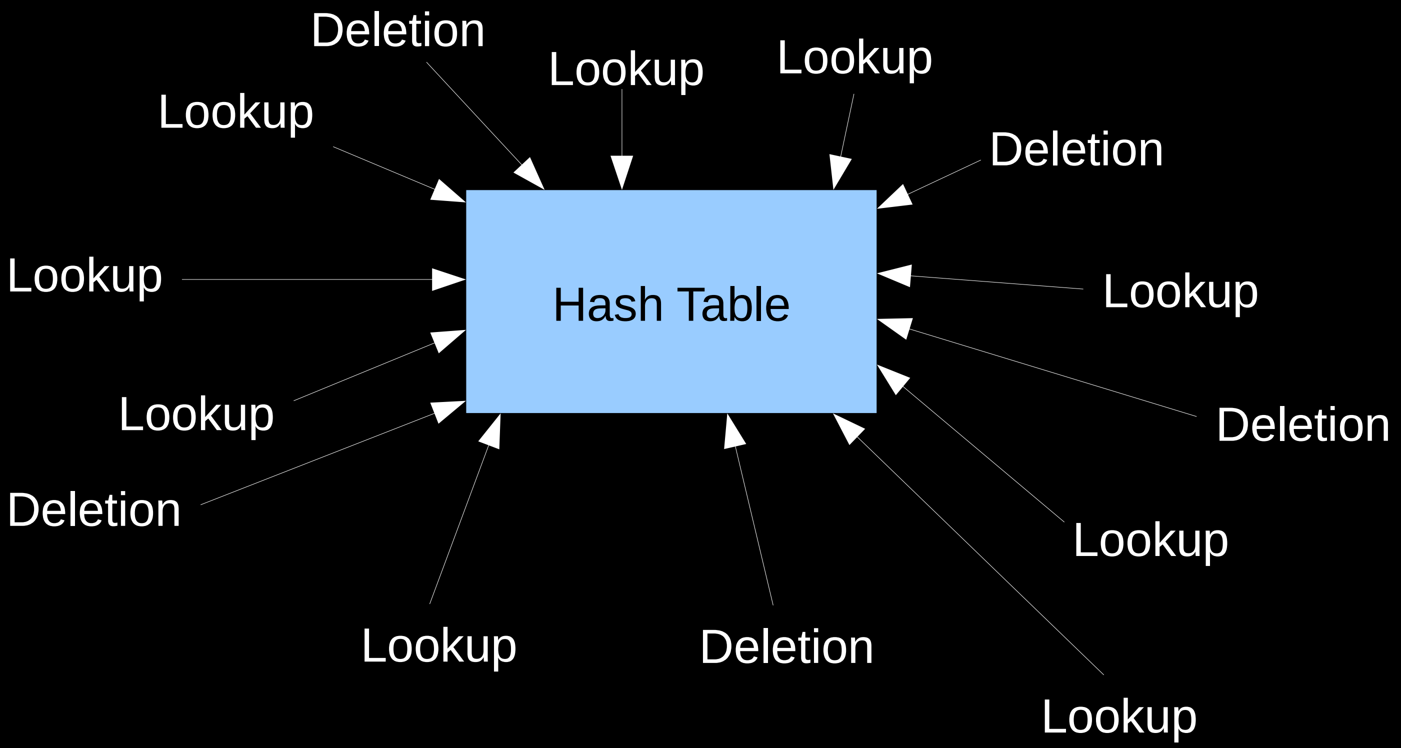
Use Non-Blocking Synchronization!

- Big issue: Lookups run concurrently with deletions
  - Bad form for a lookup to hand back a pointer to free memory



Use Non-Blocking Synchronization!

- Big issue: Lookups run concurrently with deletions
  - Bad form for a lookup to hand back a pointer to free memory
  - Results in lookups writing to shared memory, usually atomically



Performance of Synchronization Mechanisms

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

And these are best-case values!!! (Why?)

Why All These Low-Level Details???

Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?

Why All These Low-Level Details???

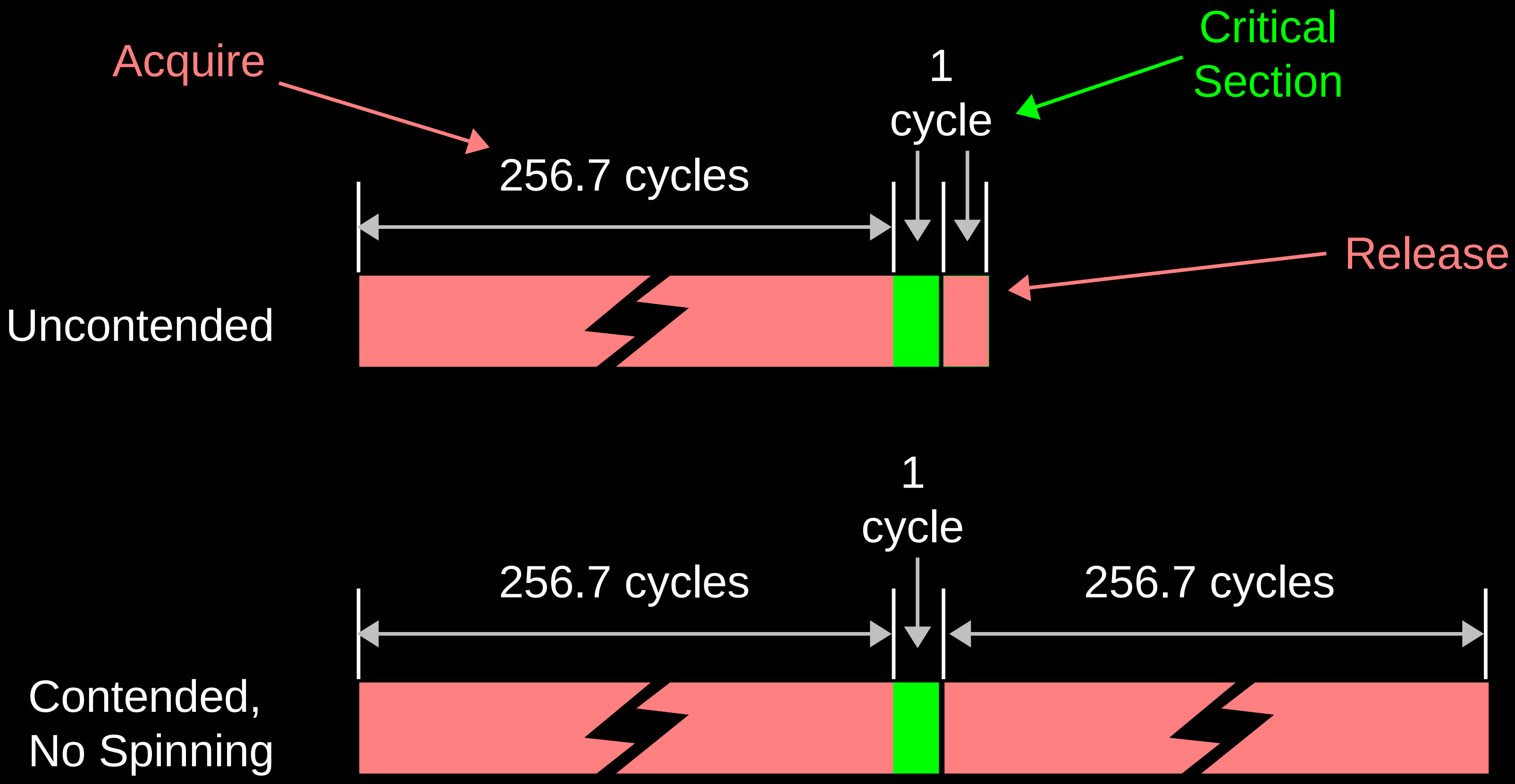
- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?

- So why trust algorithms from someone ignorant of the properties of the underlying hardware???

But What Do The Operation Timings Really Mean???

But What Do The Operation Timings Really Mean???

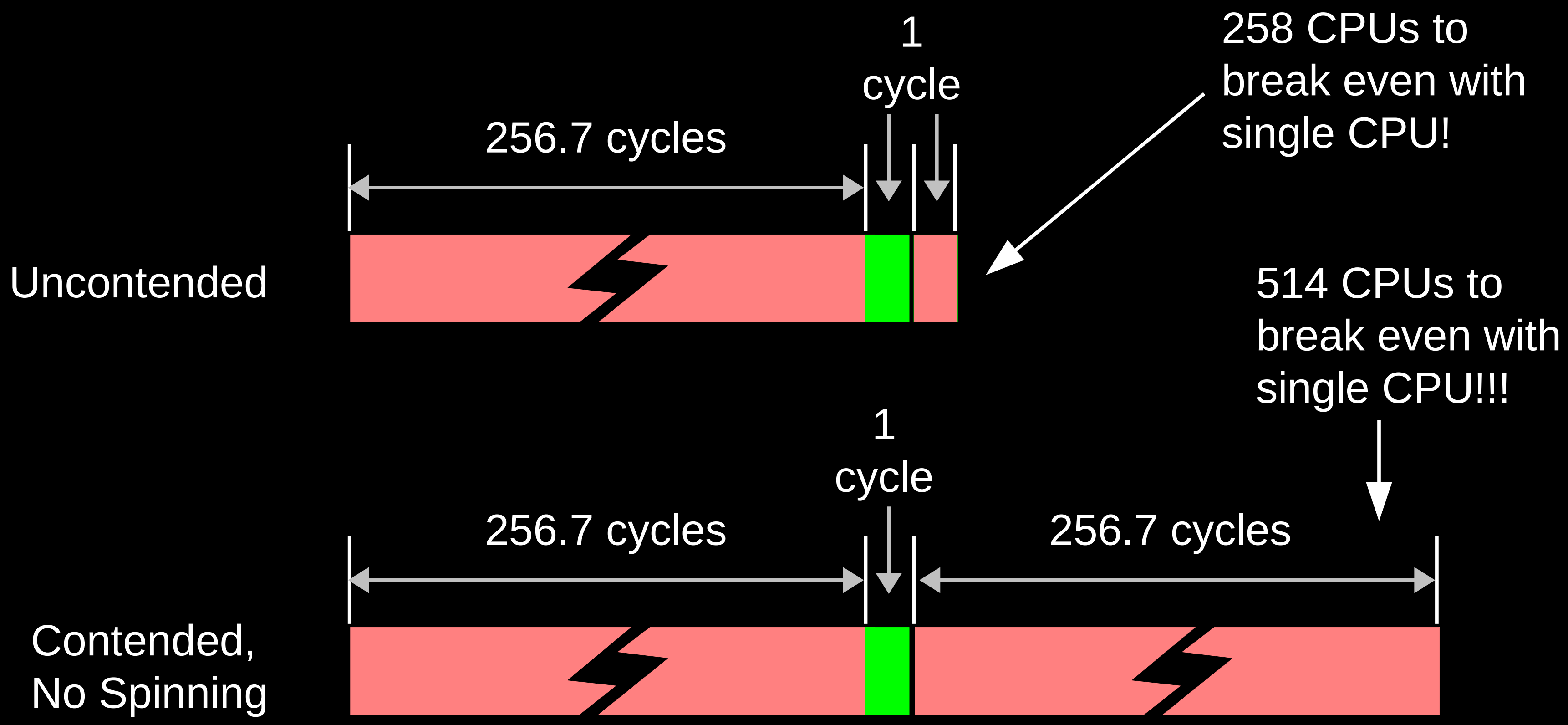
- Single-instruction critical sections protected by data locking



So, what *does* this mean?

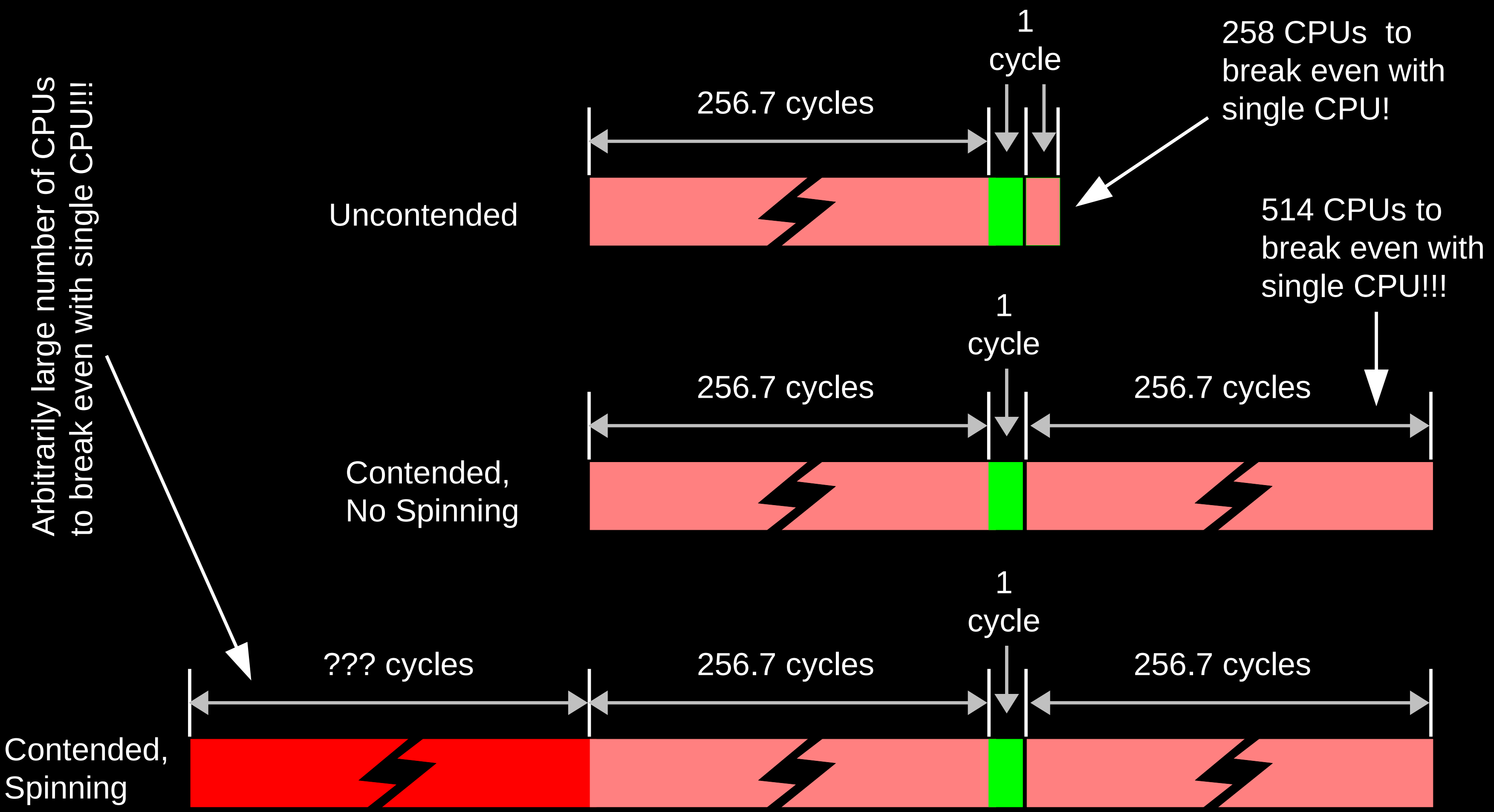
But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by data locking



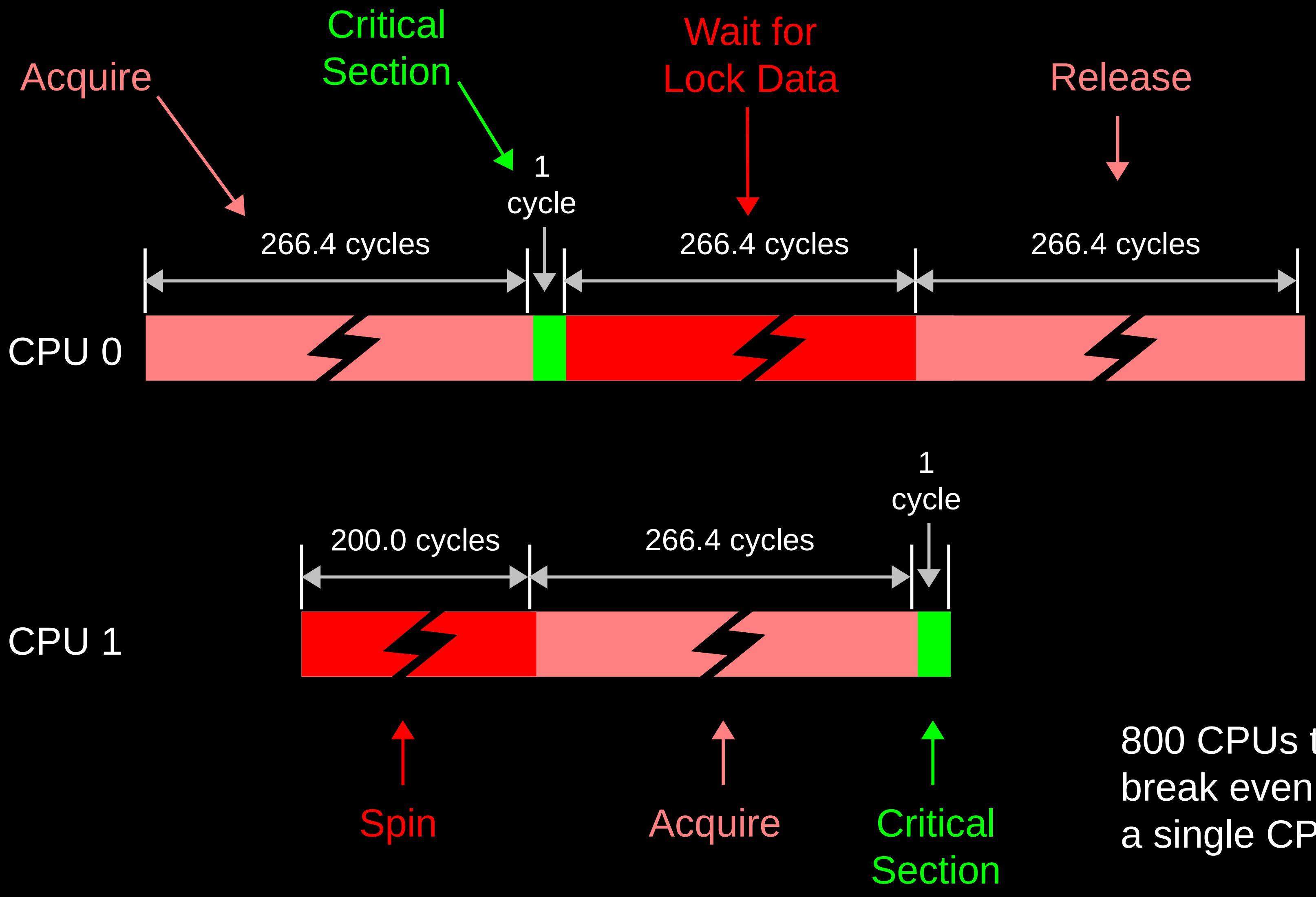
But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by data locking


1 cycle
256.7 cycles
258 CPUs to break even with single CPU!
Uncontended
514 CPUs to break even with single CPU!!!
1 cycle
256.7 cycles
256.7 cycles
Contended, No Spinning
Arbitrarily large number of CPUs to break even with single CPU!!!
1 cycle
??? cycles
256.7 cycles
256.7 cycles
Contended, Spinning

Reader-Writer Locks Are Even Worse!

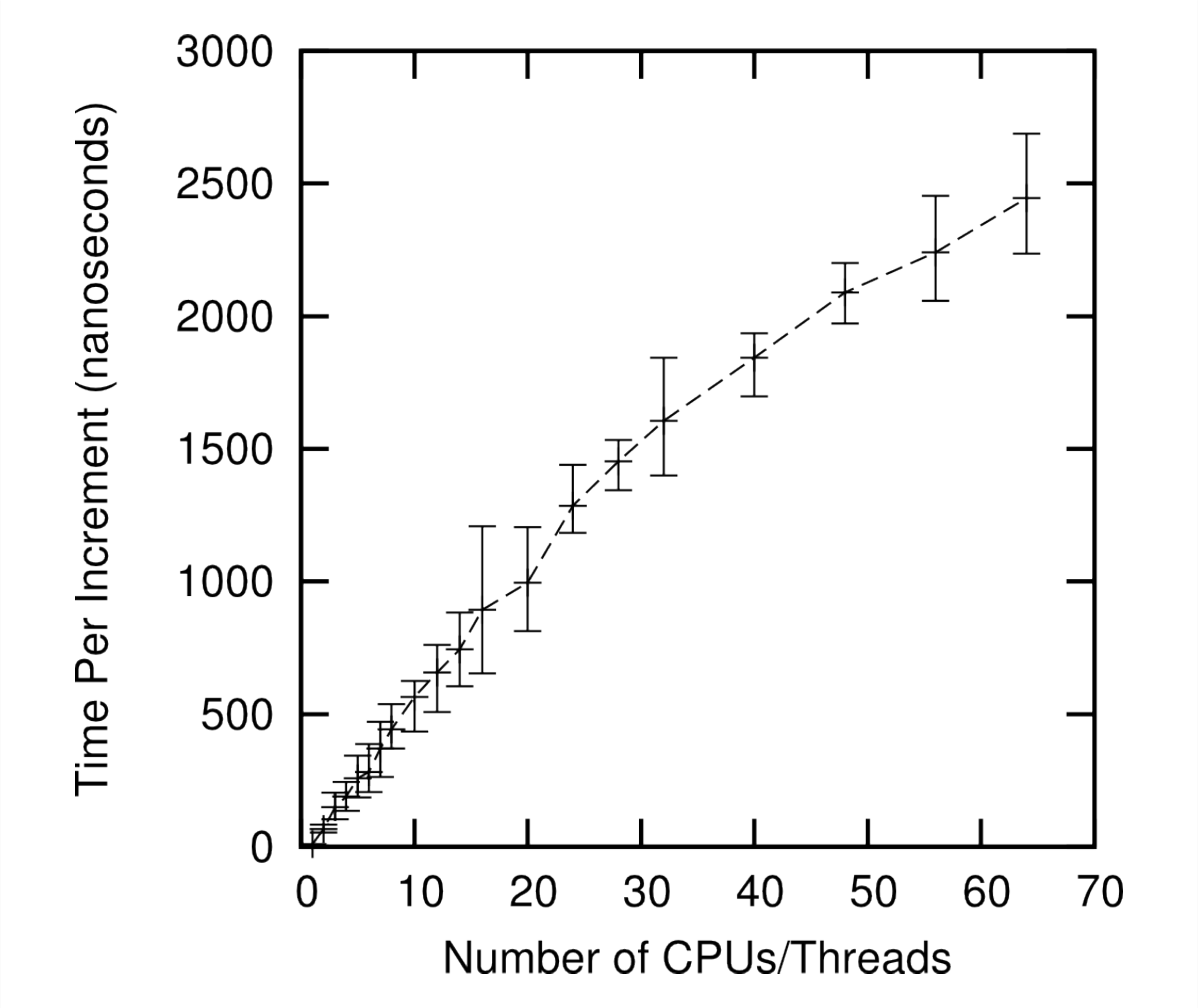
Reader-Writer Locks Are Even Worse!



800 CPUs to break even with a single CPU!!!

But What About Scaling With Atomic Operations? Non-Blocking Synchronization For The Win!!!

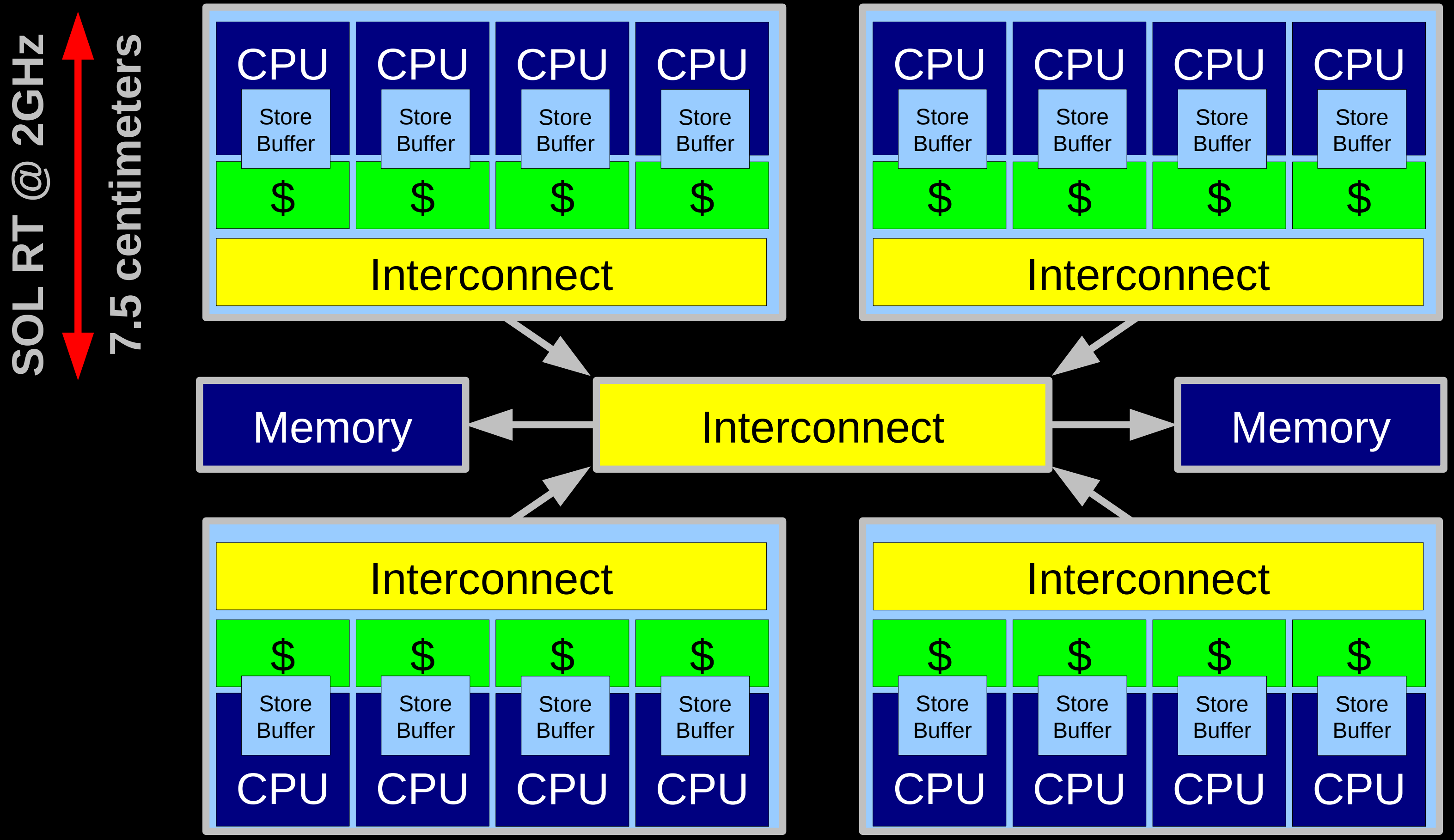
If You Think Single Atomic is Expensive, Try Lots!!!



2GHz Intel Xeon Westmere-EX

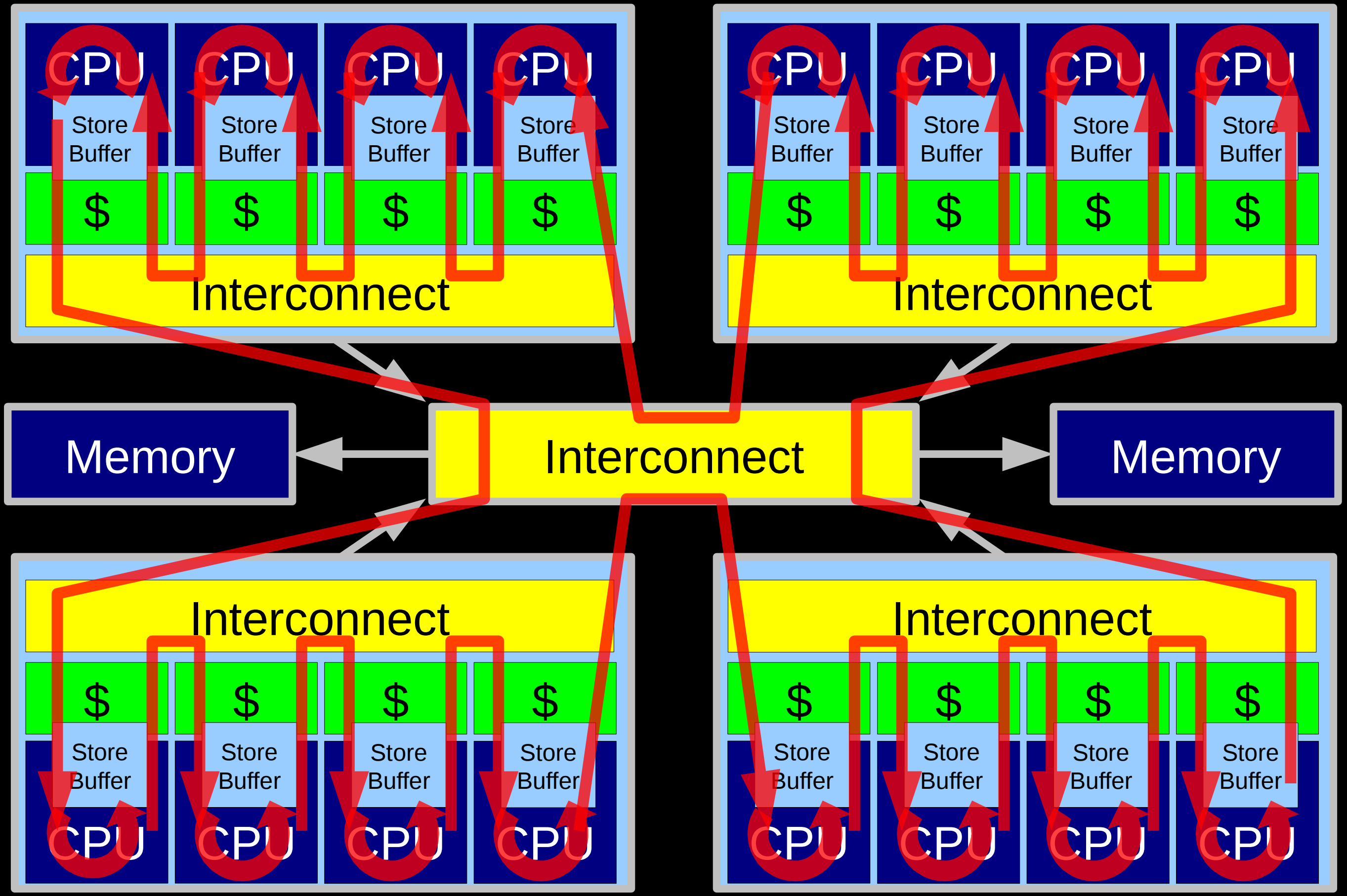
Why So Slow???

System Hardware Structure and Laws of Physics



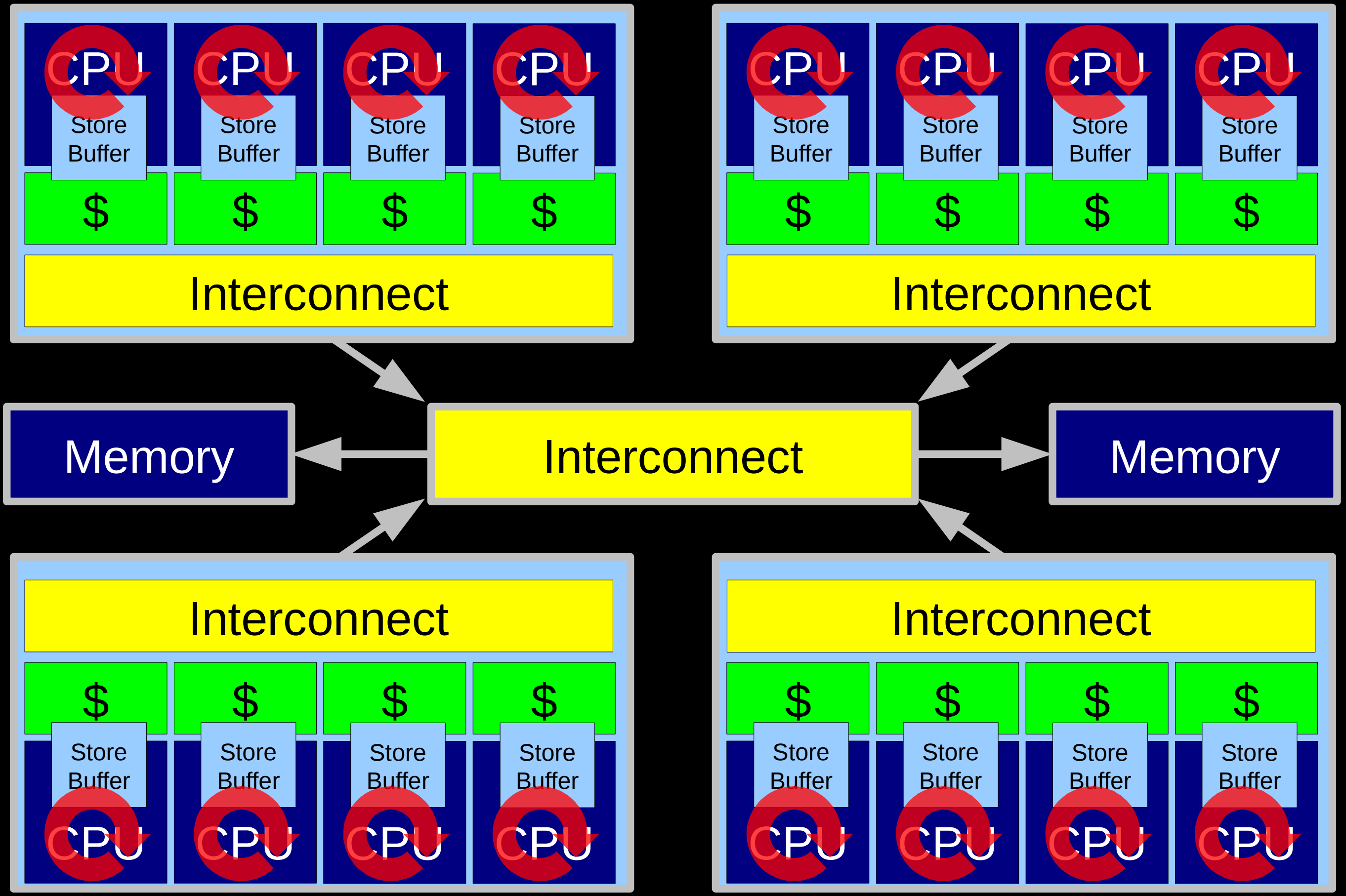
Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???

Atomic Increment of Global Variable



Lots and Lots of Latency!!!

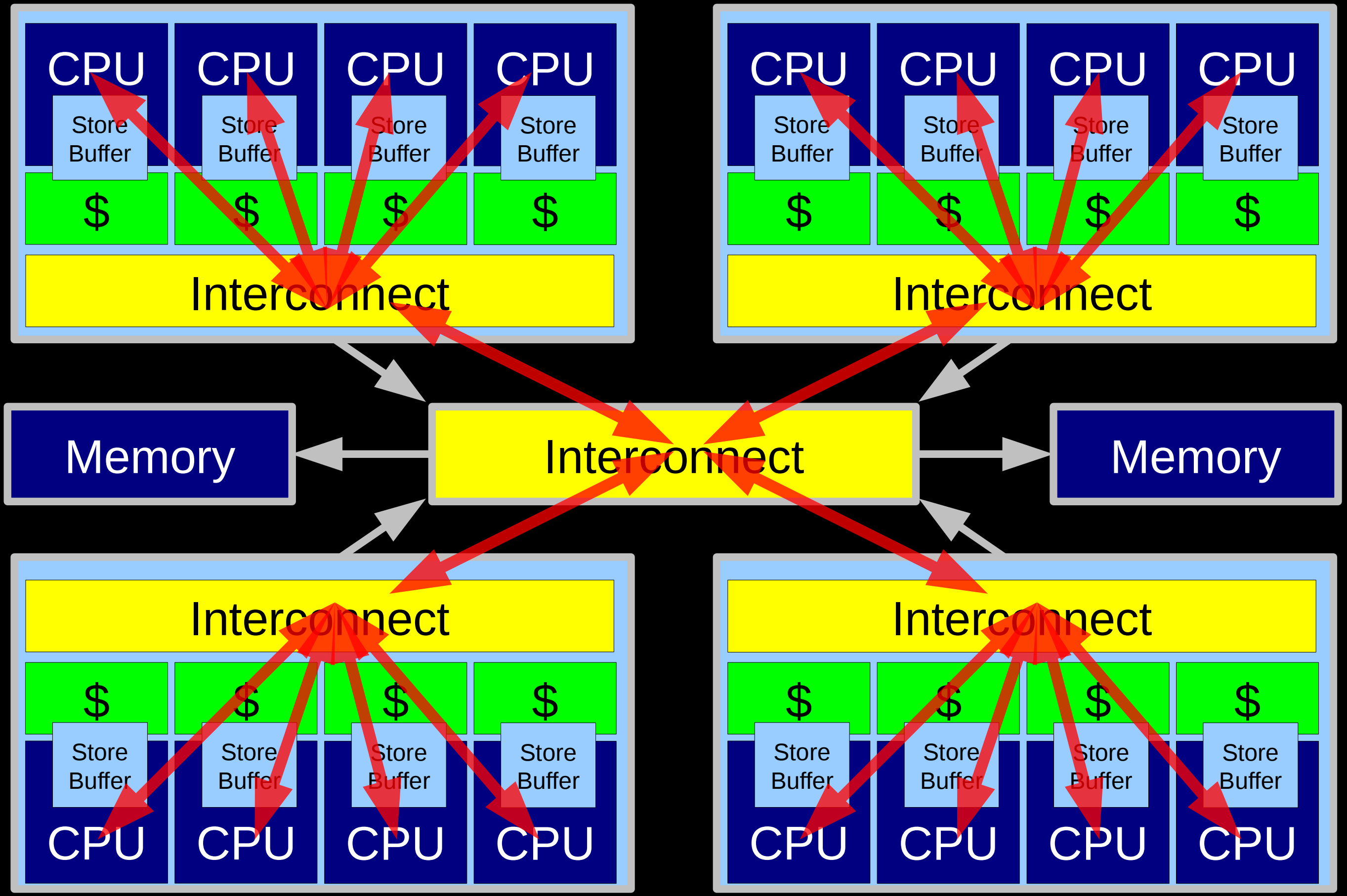
Atomic Increment of Per-CPU Counter



Little Latency, Lots of Increments at Core Clock Rate

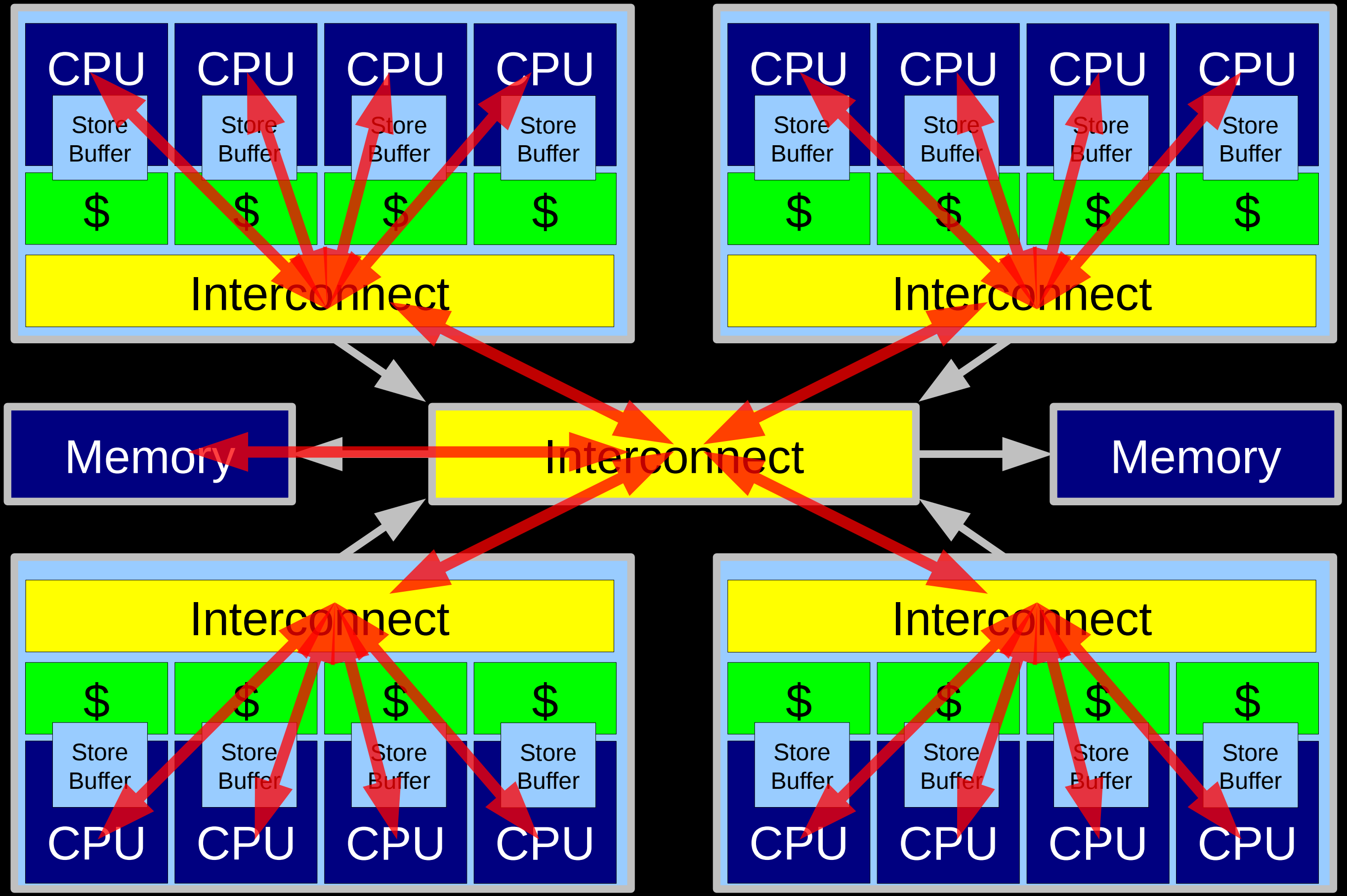
Can't The Hardware Do Better Than This???

HW-Assist Atomic Increment of Global Variable



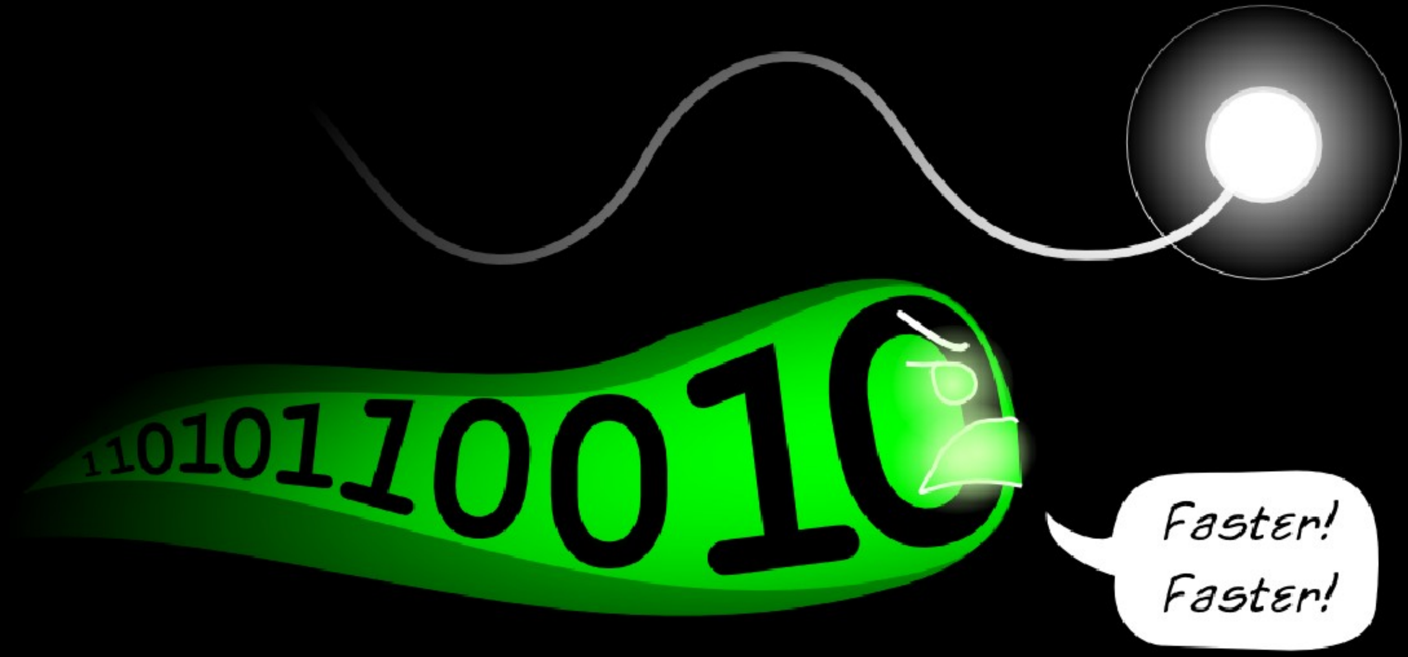
SGI systems used this approach in the 1990s, expect modern CPUs to optimize.
Still not as good as per-CPU counters.

HW-Assist Atomic Increment of Global Variable



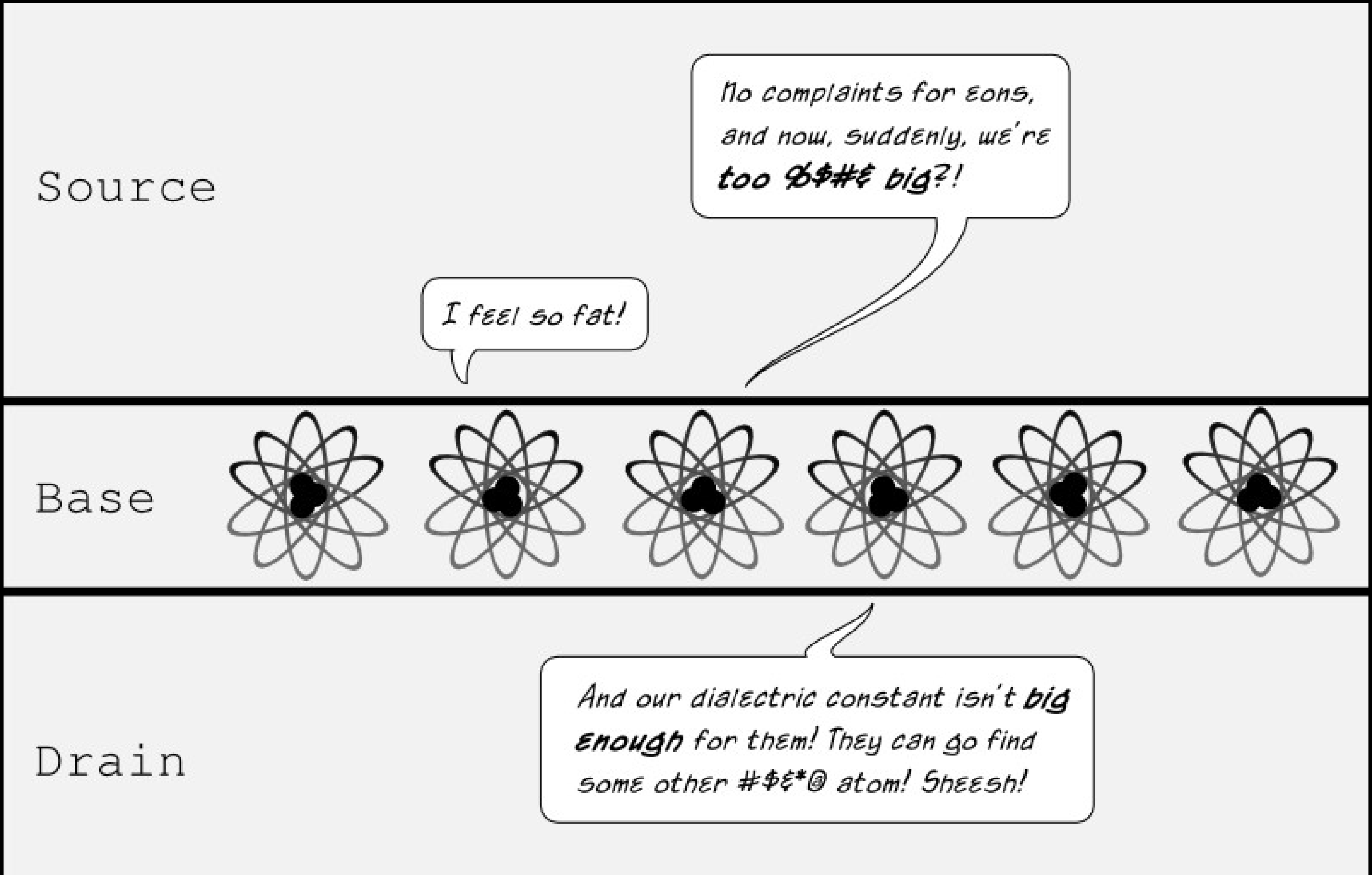
Put an ALU near memory to avoid slowdowns due to latency.
Still not as good as per-CPU counters.

Problem With Physics #1: Finite Speed of Light



(c) 2012 Melissa Broussard, Creative Commons Share-Alike

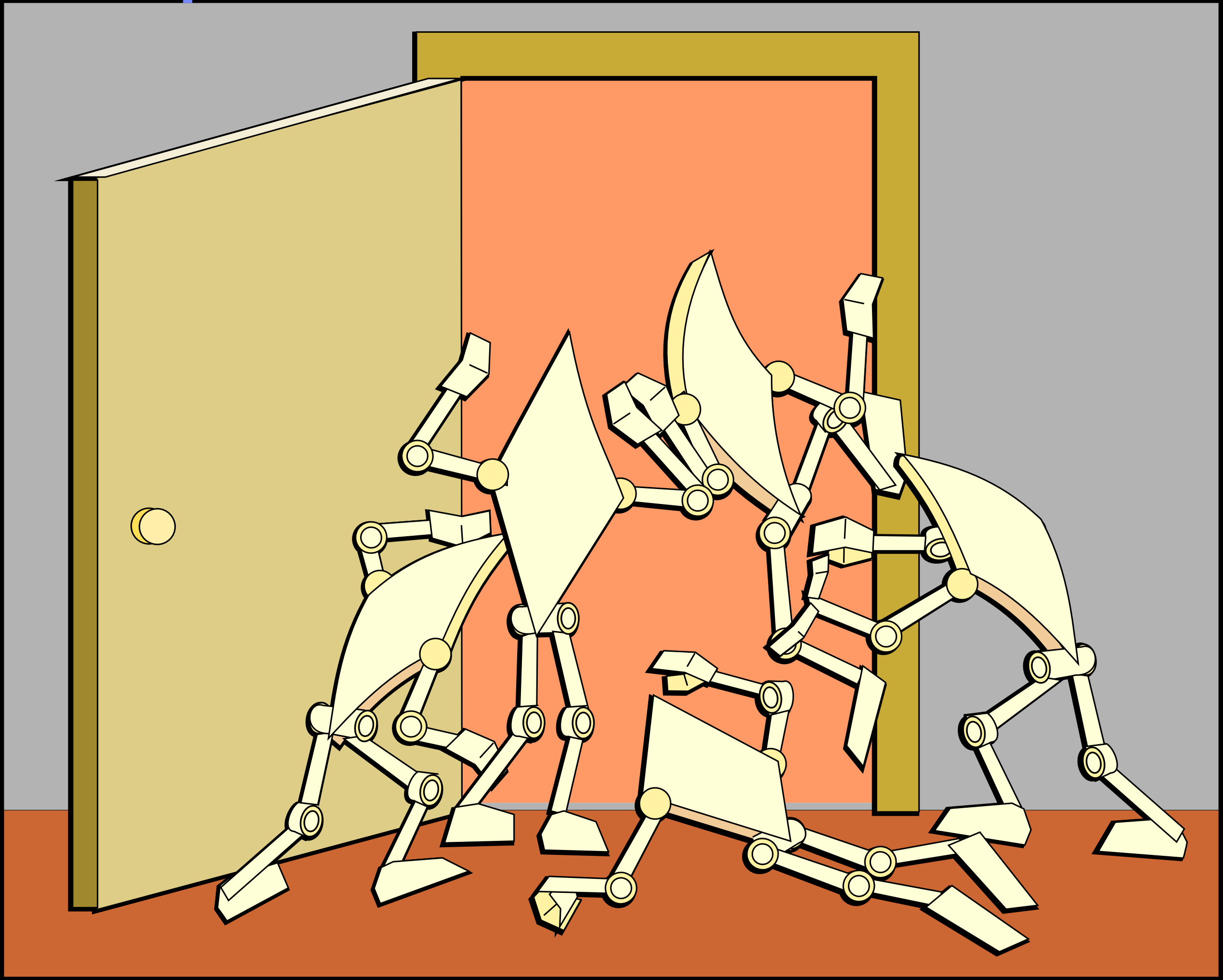
Problem With Physics #2: Atomic Nature of Matter


Source
No complaints for eons, and now, suddenly, we're too %$#¢ big?!
I feel so fat!
Base
Drain
And our dialectric constant isn't big enough for them! They can go find some other #$¢*@ atom! Sheesh!

(c) 2012 Melissa Broussard, Creative Commons Share-Alike

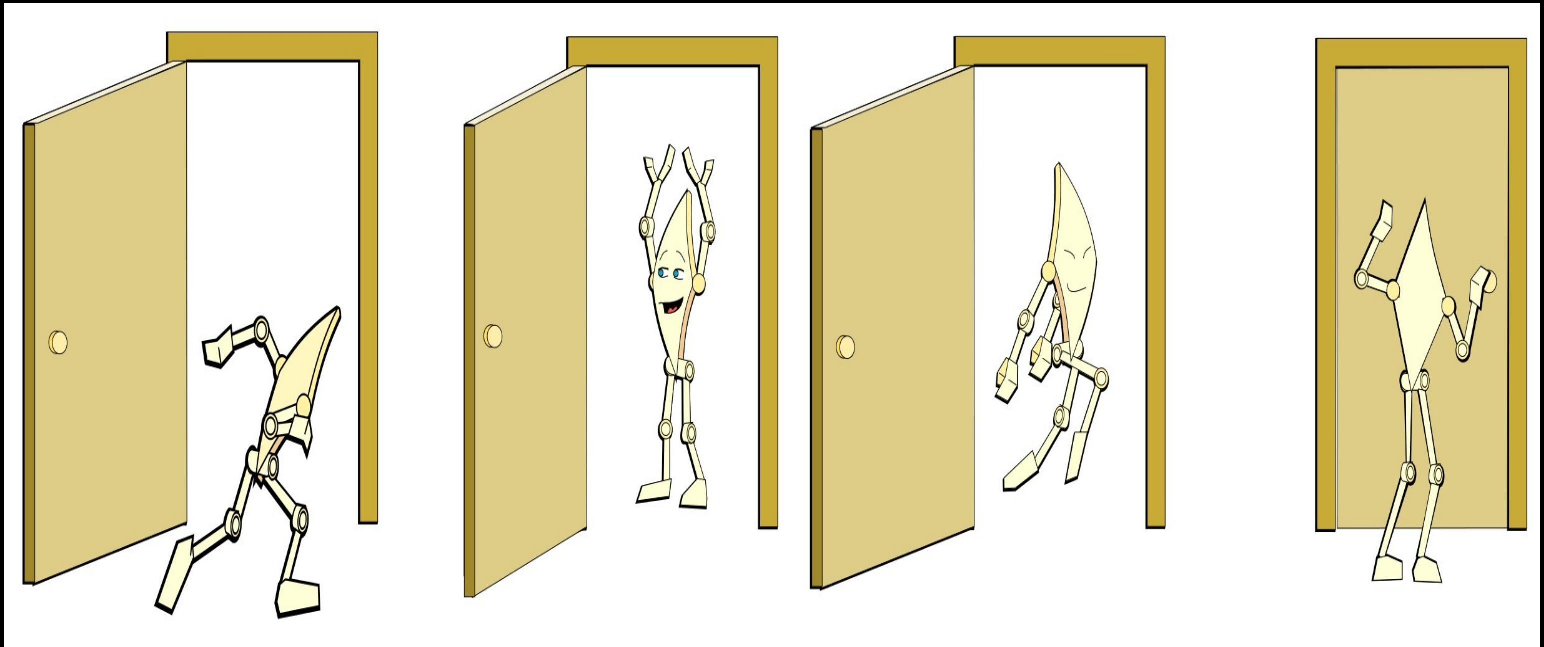
How Can Software Live With This Hardware???

Design Principle: Avoid Bottlenecks



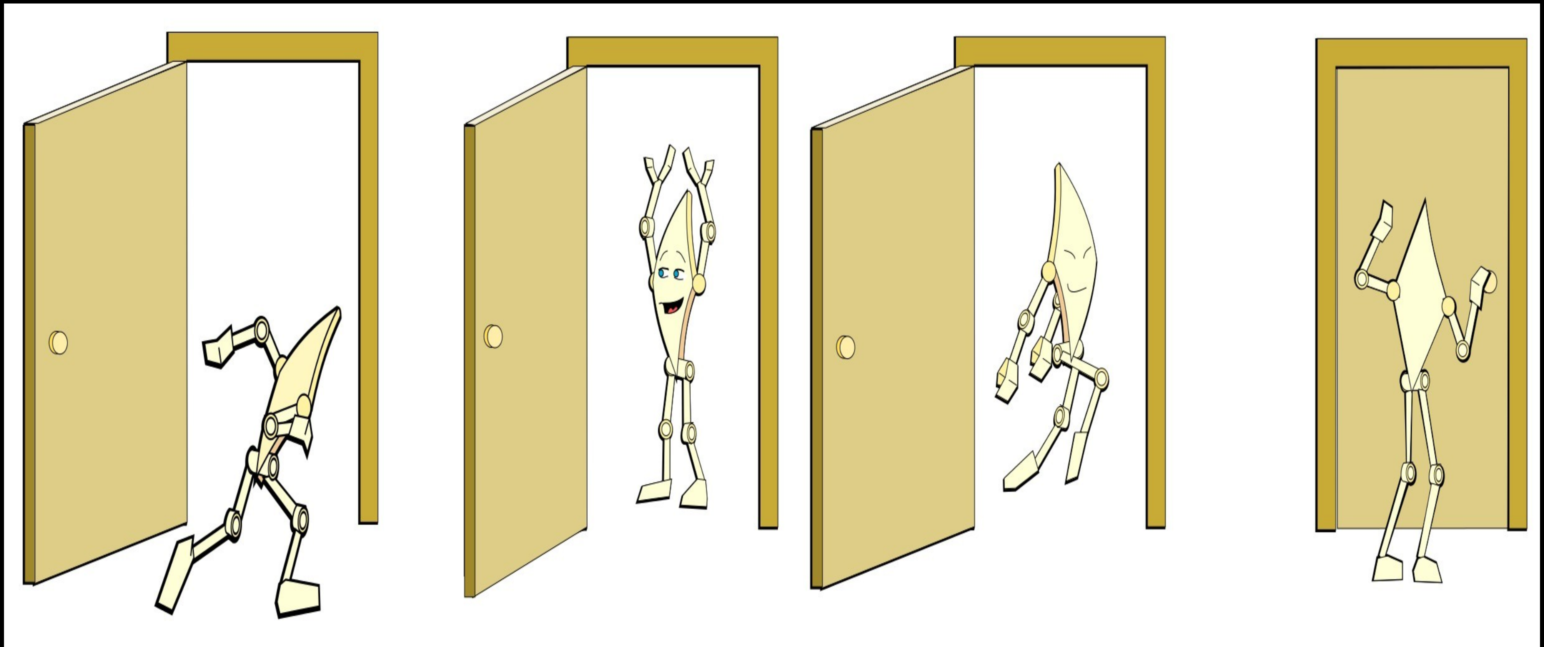
Only one of something: bad for performance and scalability.
Also typically results in high complexity.

Design Principle: Avoid Bottlenecks



Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs.
Hazard pointers uses this trick with reference counting.

Design Principle: Avoid Bottlenecks



Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs.
But NUMA effects defeated this for per-bucket locking!!!

Design Principle: Avoid Expensive Operations

Need to be here!
(Partitioning/RCU/hazptr)

But can't always!

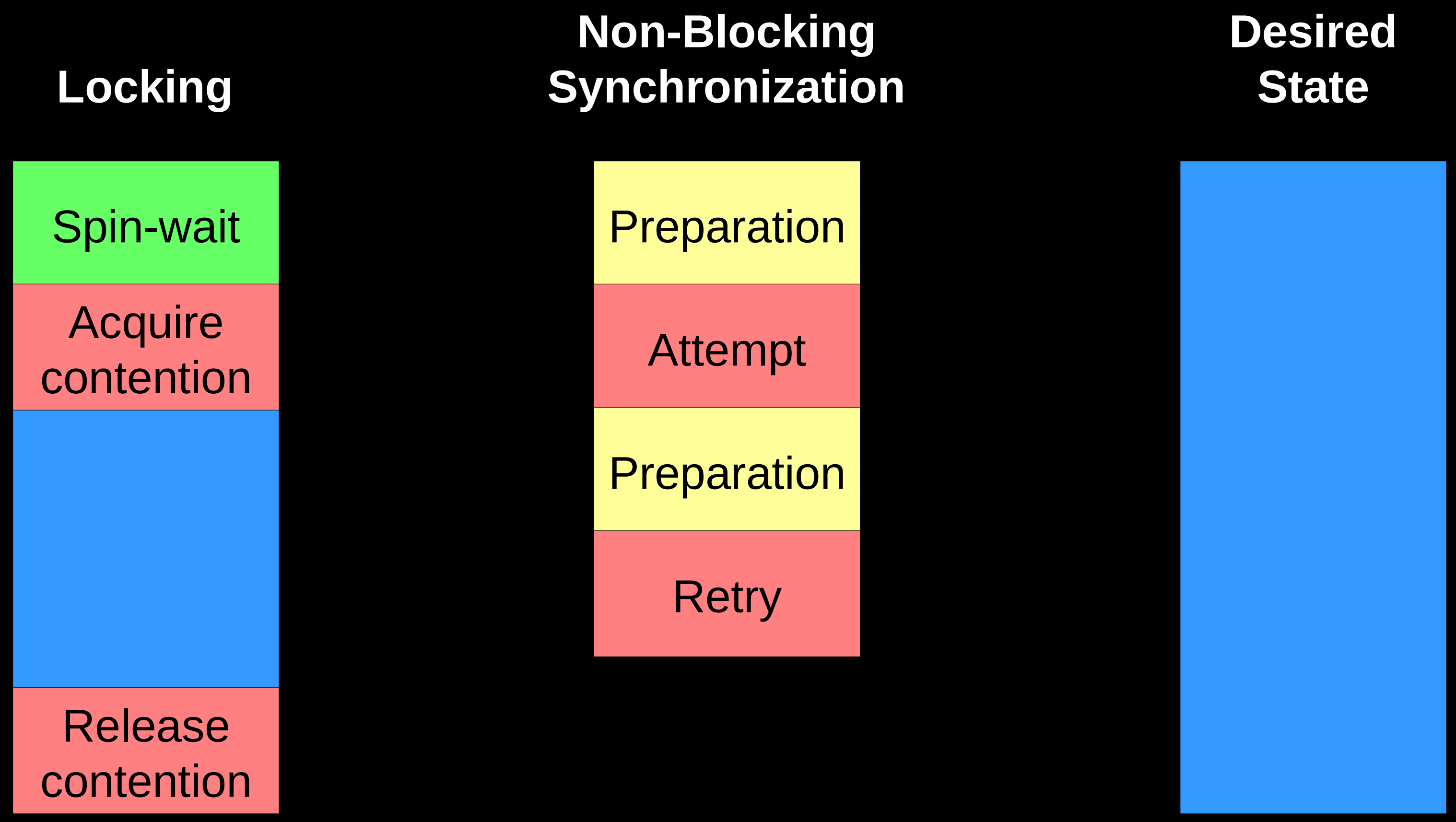
16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

Design Principle: Avoid Contention



Locking

Spin-wait

Acquire contention

Release contention

Non-Blocking Synchronization

Preparation

Attempt

Preparation

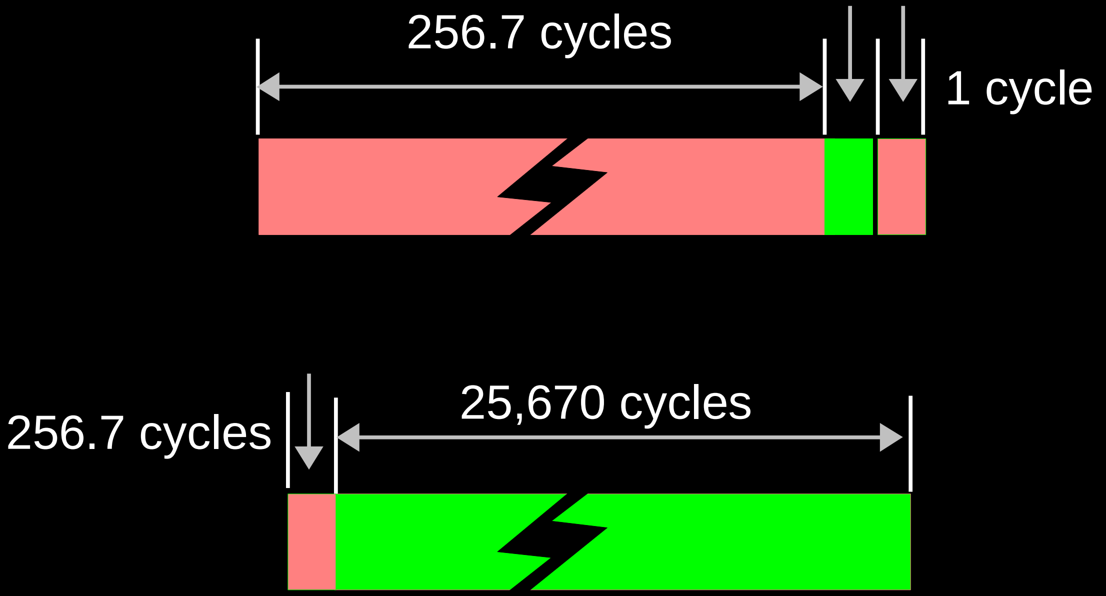
Retry

Desired State

Simple **non-blocking synchronization does very well**

Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket CAS costs about 260 cycles
  - So instead of a single-cycle critical section, have a 26000-cycle critical section, reducing synchronization overhead to about 1%



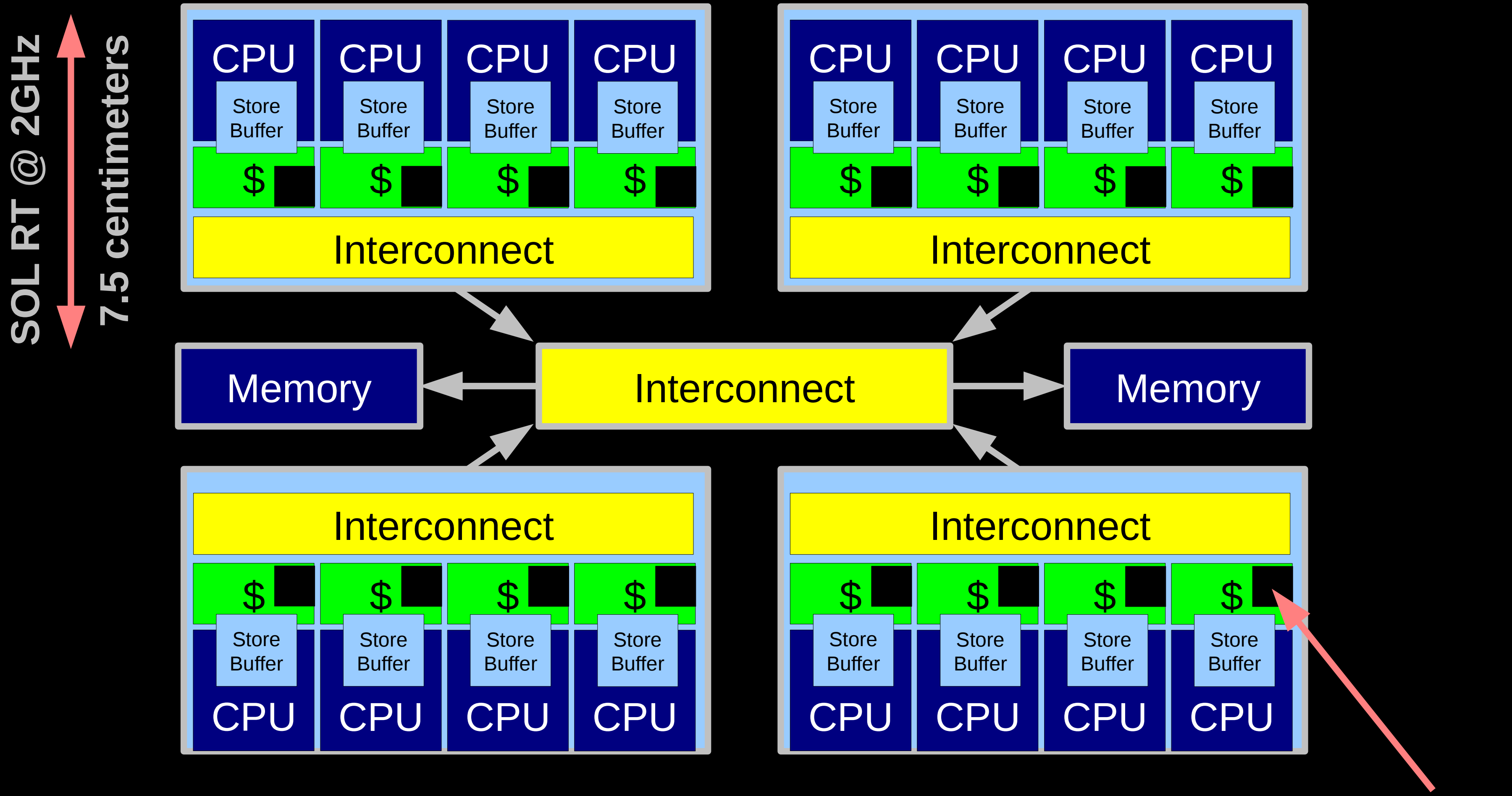
99.6% overhead

0.99% overhead

Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket CAS costs about 260 cycles
  - So instead of a single-cycle critical section, have a 26000-cycle critical section, reducing synchronization overhead to about 1%
- Of course, we also need to keep contention low, which usually means we want short critical sections
  - Resolve this by applying parallelism at as high a level as possible
  - **Parallelize entire applications rather than low-level algorithms!**
- This does not work for Schrödinger: The overhead of hash-table operations is too low
  - Which is precisely why we selected hash tables in the first place!!!

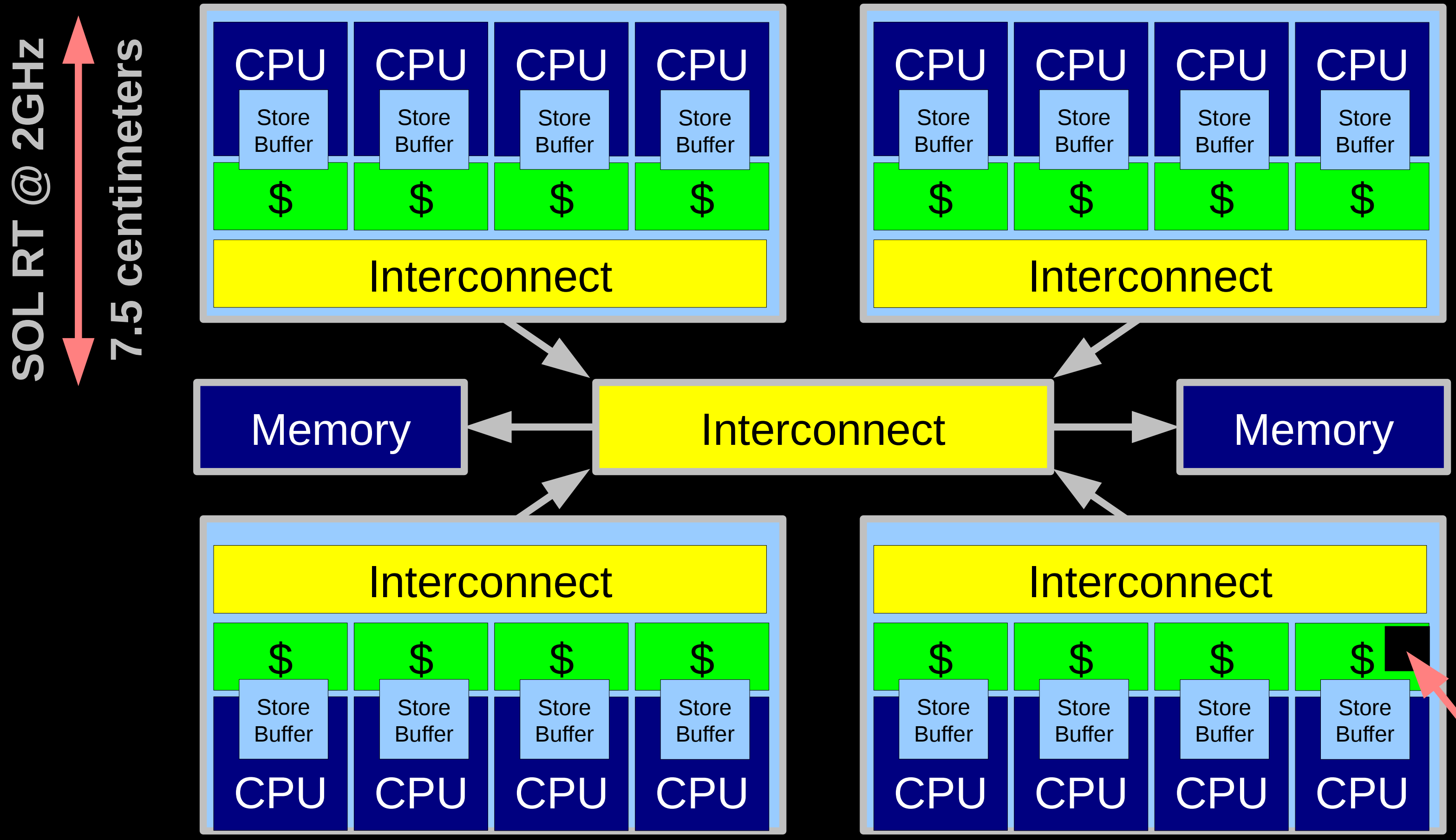
Design Principle: Leverage Read-Mostly Situations



Read-only data remains replicated in all caches

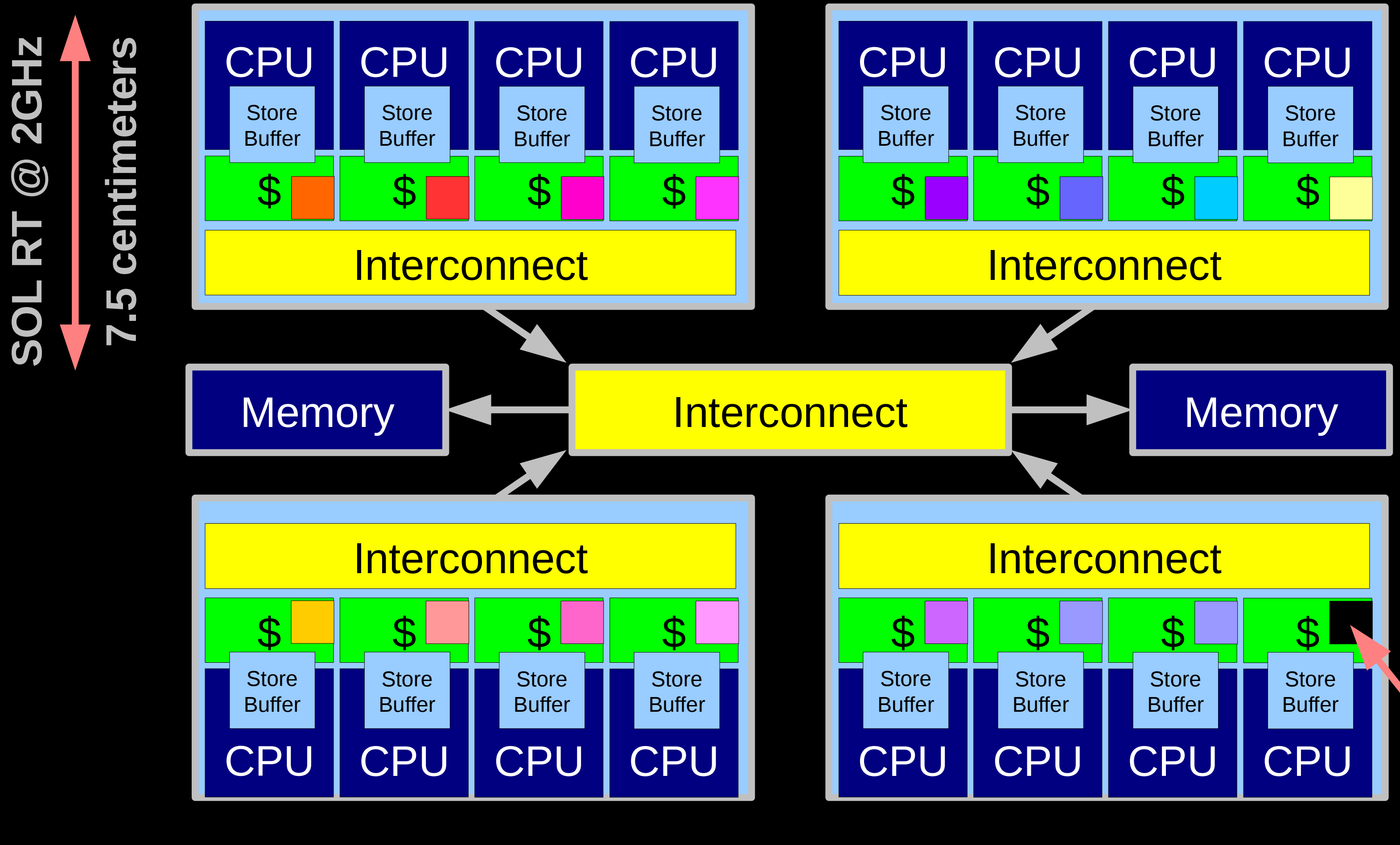
Read-mostly access dodges the laws of physics!!!

Updates Hit Hard By Unforgiving Laws of Physics



Read-only data remains replicated in all caches, but each update destroys other replicas!

Design Principle: Leverage Locality!!!



Each CPU operates on its own "shard" of the data, preserving cache locality and performance

Updates: Just Say “No”???

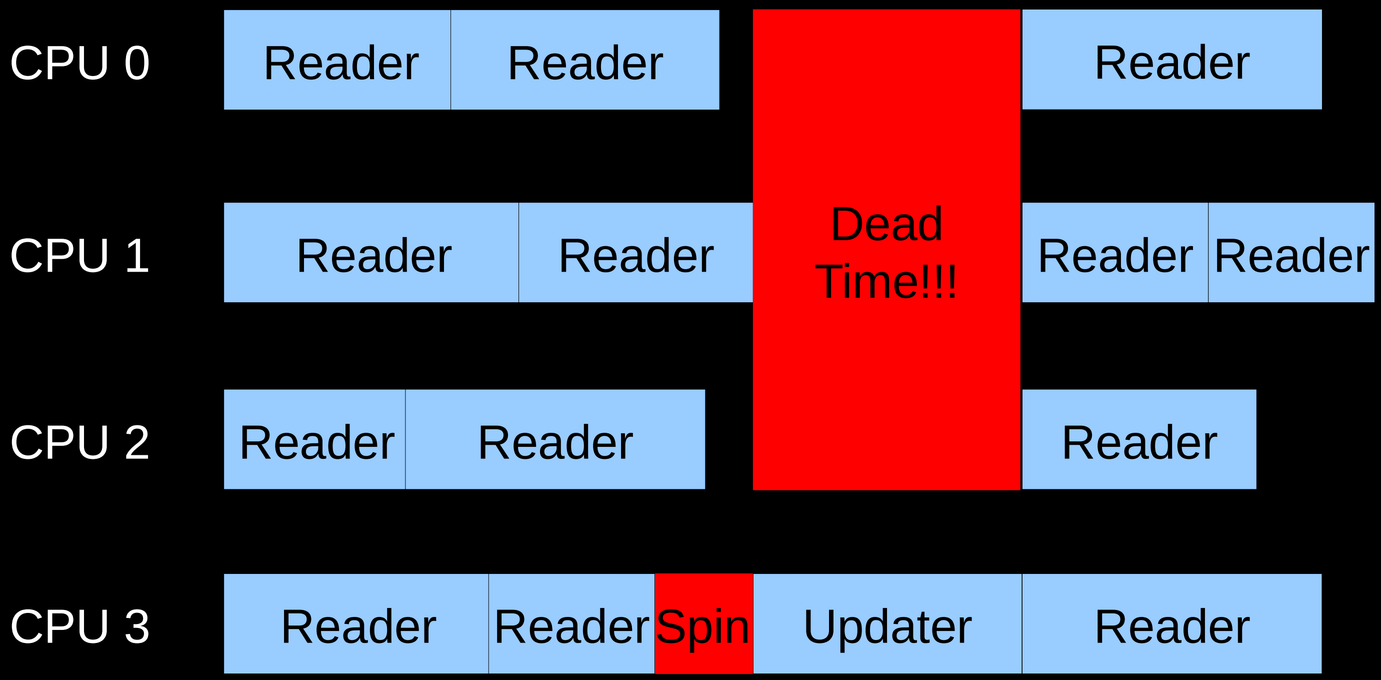
- “Doing updates is slow and non-scalable!”
- “Then don't do updates!”

Updates: Just Say “No”???

- “Doing updates is slow and non-scalable!”
- “Then don't do updates!”

OK, OK, don't do *unnecessary* updates!!!
For example, read-only traversal to update location

Design Principle: Avoid Mutual Exclusion!!!



CPU 0 Reader Reader Reader

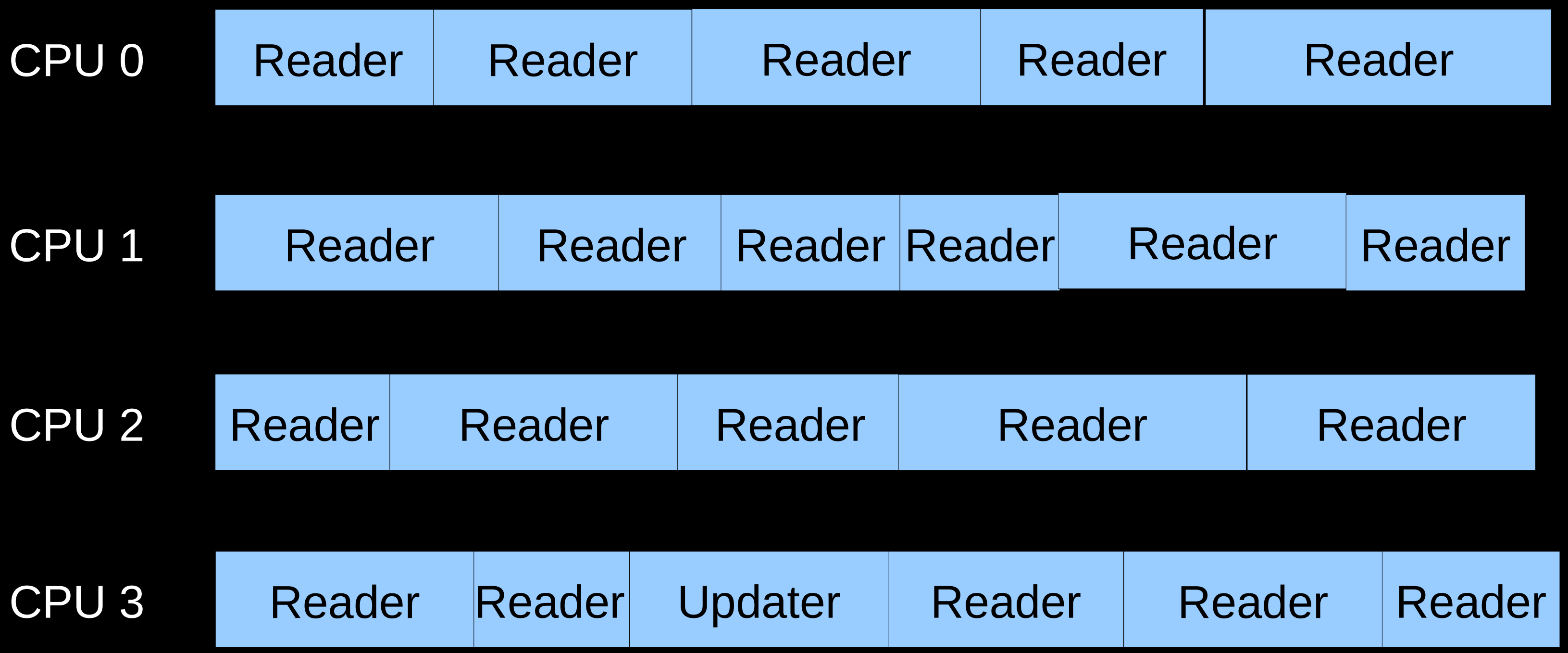
CPU 1 Reader Reader Reader Reader

Dead Time!!!

CPU 2 Reader Reader Reader

CPU 3 Reader Reader Spin Updater Reader

Design Principle: Avoiding Mutual Exclusion



CPU 0: Reader Reader Reader Reader Reader

CPU 1: Reader Reader Reader Reader Reader Reader

CPU 2: Reader Reader Reader Reader Reader

CPU 3: Reader Reader Updater Reader Reader Reader

No Dead Time!

But How Can This Possibly Be Implemented???

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Advantages:
- Disadvantage:

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Advantages: Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency
- Disadvantage: How can something that does not affect machine state possibly be used as a synchronization primitive???

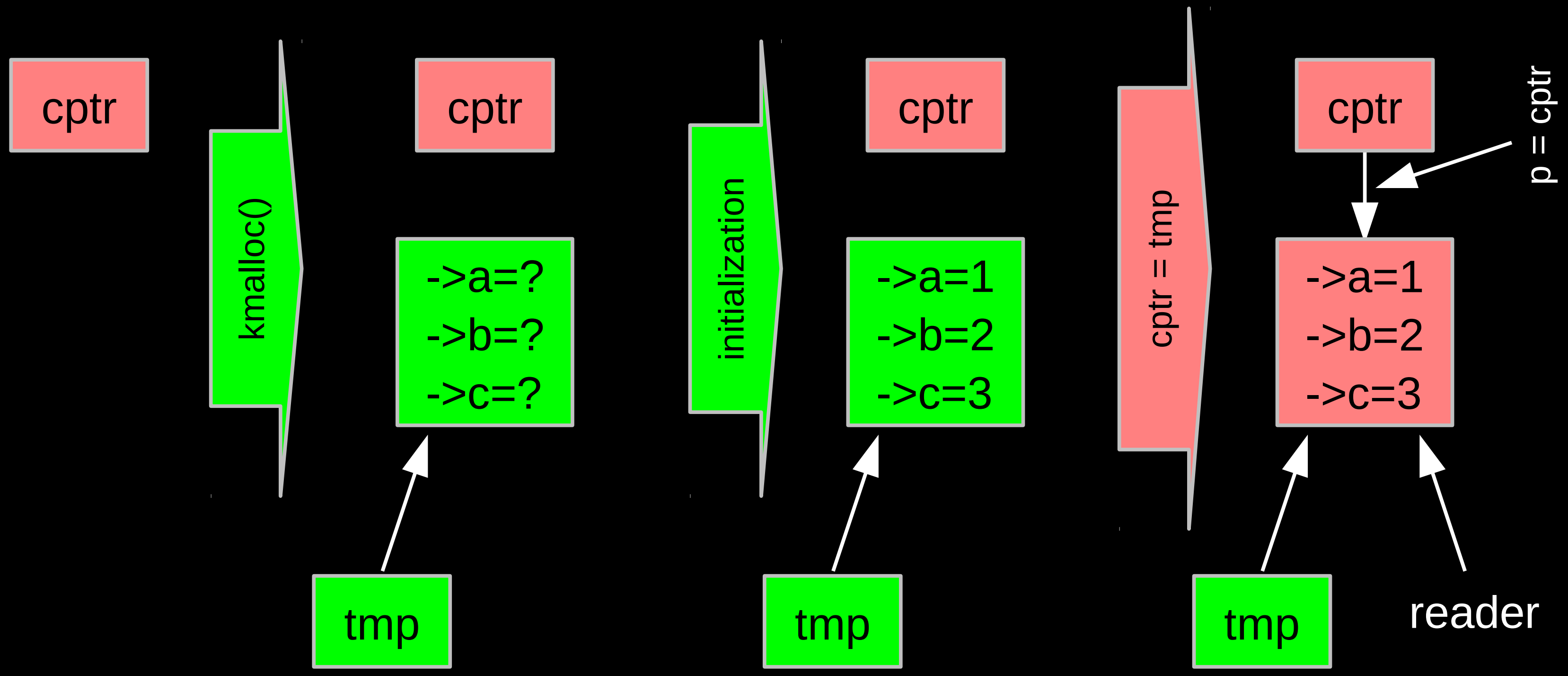
What Is RCU?

- Publishing of new data
- Subscribing to the current version of data
- Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  - Each *reade*r continues traversing its *copy* of the data while a new *copy* might be being created concurrently by each *update*r *
    - Hence the name *read-copy update*, or RCU
  - Once all pre-existing RCU readers are done with them, old versions of the data may be discarded

* This backronym expansion provided by Jonathan Walpole

Publication of And Subscription to New Data

Key: 🟥 Dangerous for updates: all readers can access
🟨 Still dangerous for updates: pre-existing readers can access (next slide)
🟩 Safe for updates: inaccessible to all readers



Memory Ordering: Mischief From Compiler and CPU

Memory Ordering: Mischief From Compiler and CPU

- Original updater code:

```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
cptr = p;
```

- Original reader code:

```
p = cptr;
foo(p->a, p->b, p->c);
```

- Mischievous updater code:

```
p = malloc(sizeof(*p));
cptr = p;
p->a = 1;
p->b = 2;
p->c = 3;
```

- Mischievous reader code:

```
retry:
p = guess(cptr);
foo(p->a, p->b, p->c);
if (p != cptr)
    goto retry;
```

Memory Ordering: Mischief From Compiler and CPU

- Original updater code:

```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
cptr = p;
```

- Original reader code:

```
p = cptr;
foo(p->a, p->b, p->c);
```

- Mischievous updater code:

```
p = malloc(sizeof(*p));
cptr = p;
p->a = 1;
p->b = 2;
p->c = 3;
```

- Mischievous reader code:

```
retry:
p = guess(cptr);
foo(p->a, p->b, p->c);
if (p != cptr)
    goto retry;
```

But don't take *my* word for it on HW value speculation:
http://www.openvms.compaq.com/wizard/wiz_2637.html

Preventing Memory-Order Mischief

- Updater uses rcu_assign_pointer() to publish pointer:

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); /* SMP Write Memory Barrier */ \
        (p) = (v); \
})
```

- Reader uses rcu_dereference() to subscribe to pointer:

```
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- The Linux-kernel definitions are more ornate: Debugging code

Preventing Memory-Order Mischief

- “Memory-order-mischief proof” updater code:

```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
rcu_assign_pointer(cptr, p);
```

- “Memory-order-mischief proof” reader code:

```
p = rcu_dereference(cptr);
foo(p->a, p->b, p->c);
```

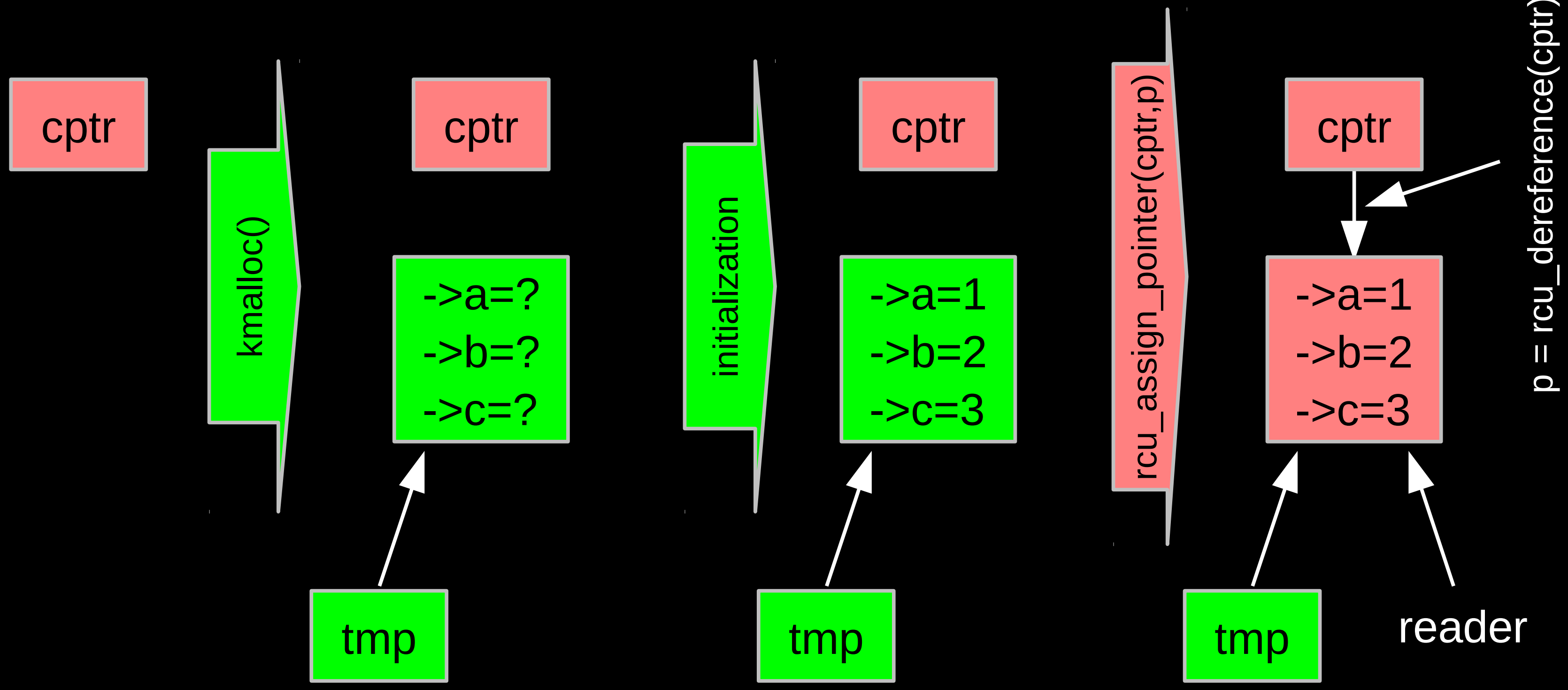
Publication of And Subscription to New Data

Key:

🟥 Dangerous for updates: all readers can access

🟨 Still dangerous for updates: pre-existing readers can access (next slide)

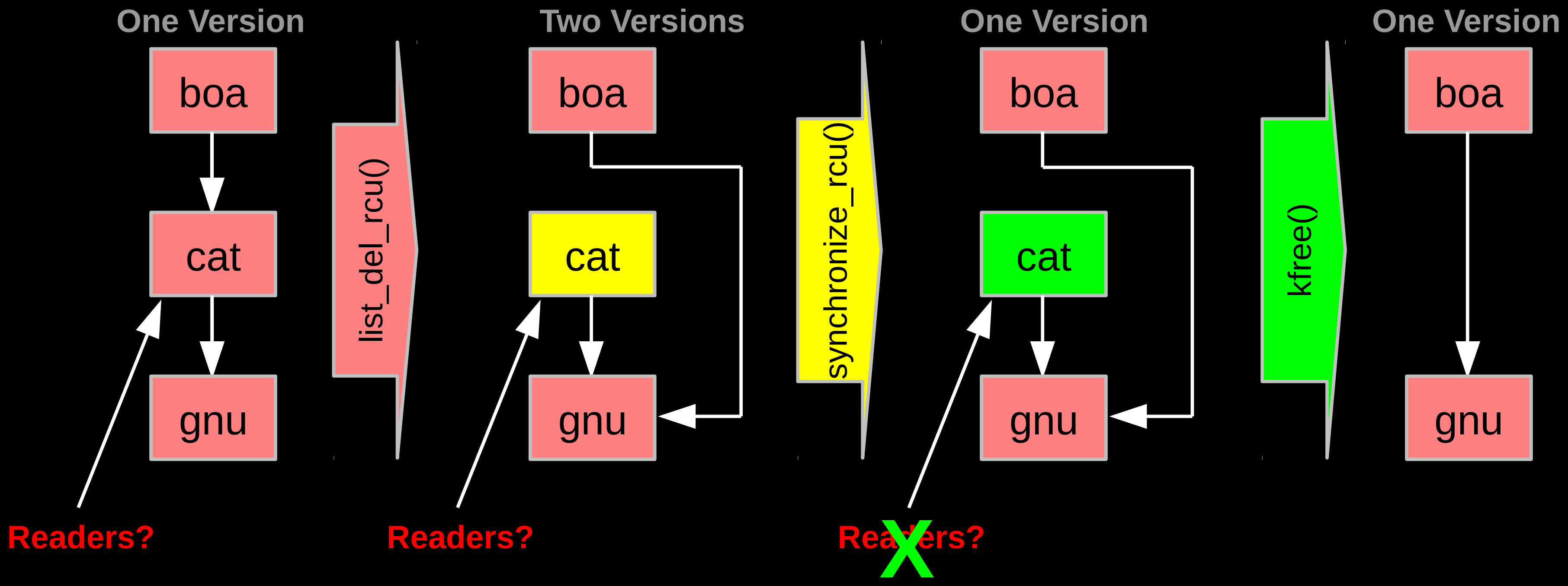
🟩 Safe for updates: inaccessible to all readers



But if all we do is add, we have a big memory leak!!!

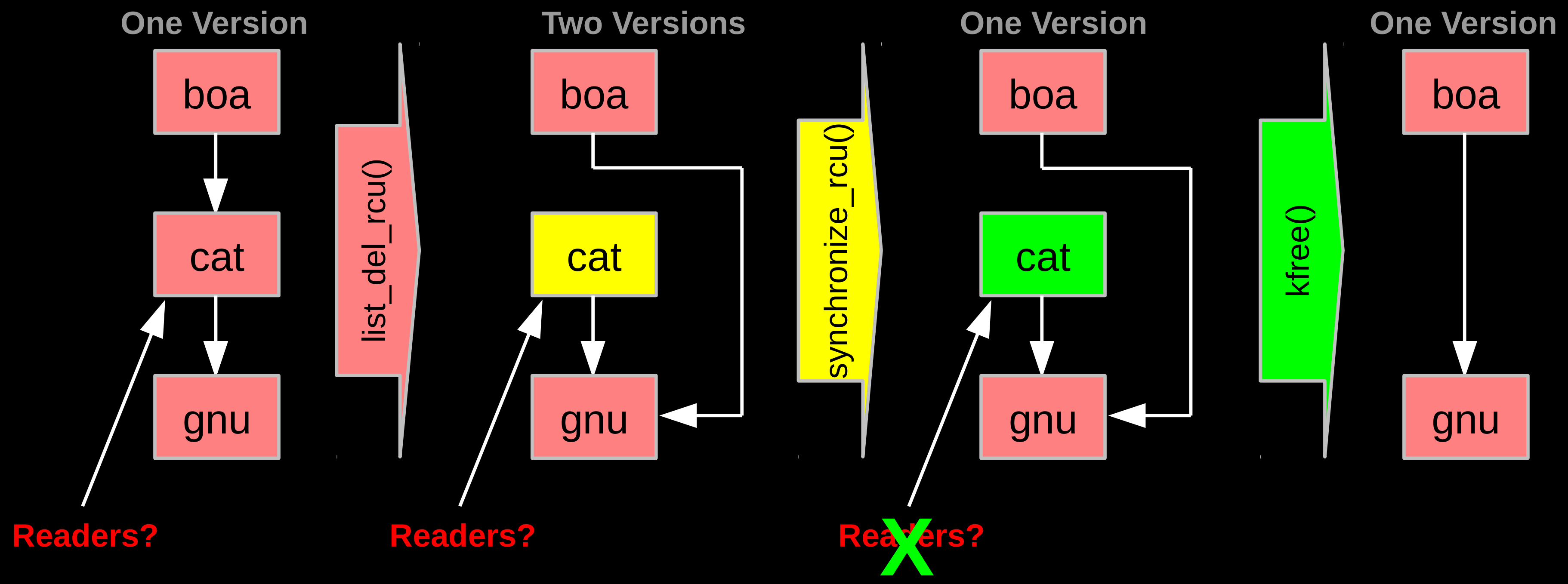
RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



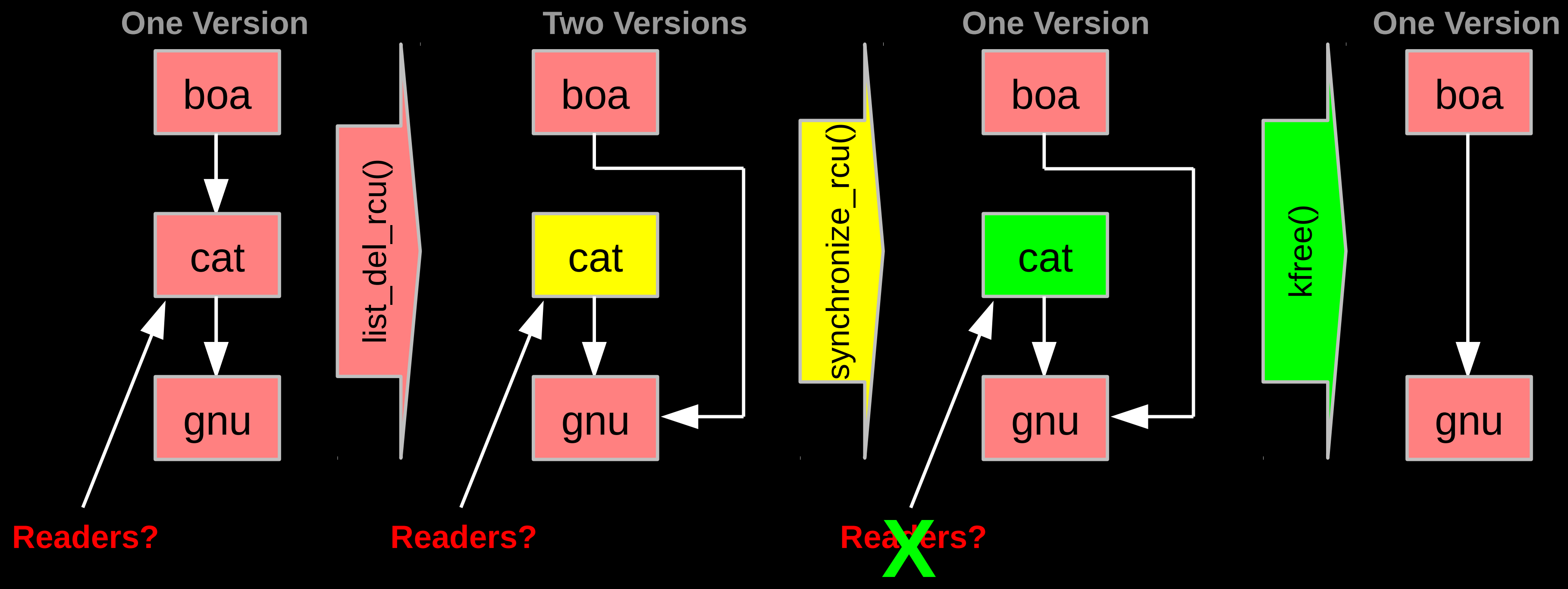
But how can software deal with two different versions simultaneously???

Two Different Versions Simultaneously???

I think the poor thing has expired
No!
Where there is a brain-wave, there is a way!

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



But if readers leave no trace in memory, how can we possibly tell when they are done???

How Can RCU Tell When Readers Are Done???

How Can RCU Tell When Readers Are Done???

That is, without re-introducing all of the overhead and latency inherent to other synchronization mechanisms...

But First, Some RCU Nomenclature

- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread was in at least one quiescent state

But First, Some RCU Nomenclature

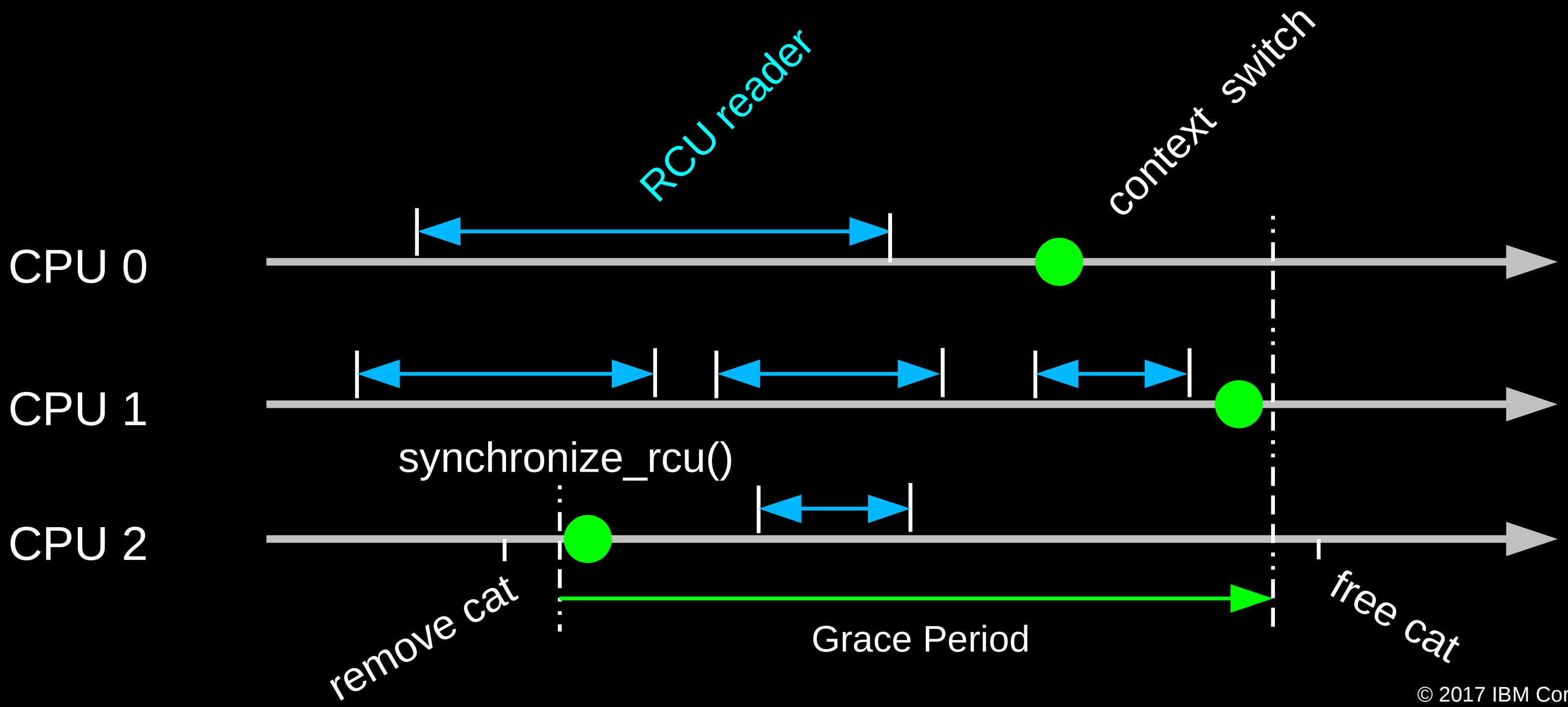
- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread was in at least one quiescent state
- OK, names are nice, but how can you possibly implement this???

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks
- CPU context switch means all that CPU's readers are done
- *Grace period* ends after all CPUs execute a context switch



Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***
- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***
- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”
- RCU is unusual is being a purely social-engineering approach
  - But RCU implementations for preemptive environments do use lightweight code in addition to social engineering

Toy Implementation of RCU: 20 Lines of Code

- Read-side primitives:

```
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- Update-side primitives

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); \
        (p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

        for_each_online_cpu(cpu)
                run_on(cpu);
}
```

Toy Implementation of RCU: 20 Lines of Code, Full Read-Side Performance!!!

- Read-side primitives:

```
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- Update-side primitives

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); \
        (p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

        for_each_online_cpu(cpu)
                run_on(cpu);
}
```

Only 9 of which are needed on sequentially consistent systems...
And some people still insist that RCU is complicated... ;-)

RCU Usage: Readers

- Pointer to RCU-protected object guaranteed to exist throughout RCU read-side critical section

```
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr);
/* *p guaranteed to exist. */
do_something_with(p);
rcu_read_unlock(); /* End critical section. */
/* *p might be freed!!! */
```

- The rcu_read_lock(), rcu_dereference() and rcu_read_unlock() primitives are very light weight
- However, updaters must take care...

RCU Usage: Updaters

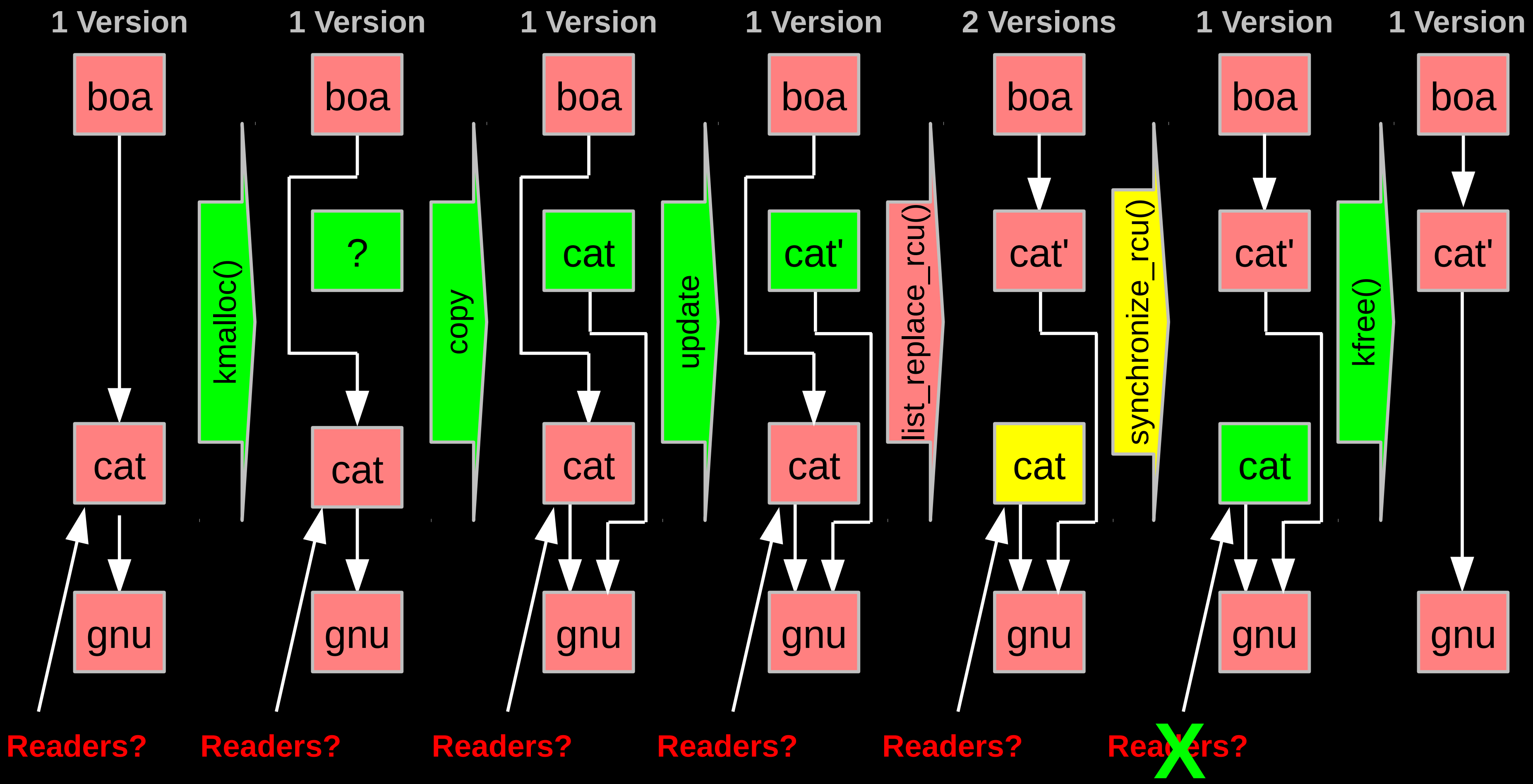
- Updaters must wait for an *RCU grace period* to elapse between making something inaccessible to readers and freeing it

```
spin_lock(&updater_lock);
q = cptr;
rcu_assign_pointer(cptr, new_p);
spin_unlock(&updater_lock);
synchronize_rcu(); /* Wait for grace period. */
kfree(q);
```

- RCU grace period waits for all pre-exiting readers to complete their RCU read-side critical sections

Complex Atomic-To-Reader Updates, Take 1

RCU Replacement Of Item In Linked List

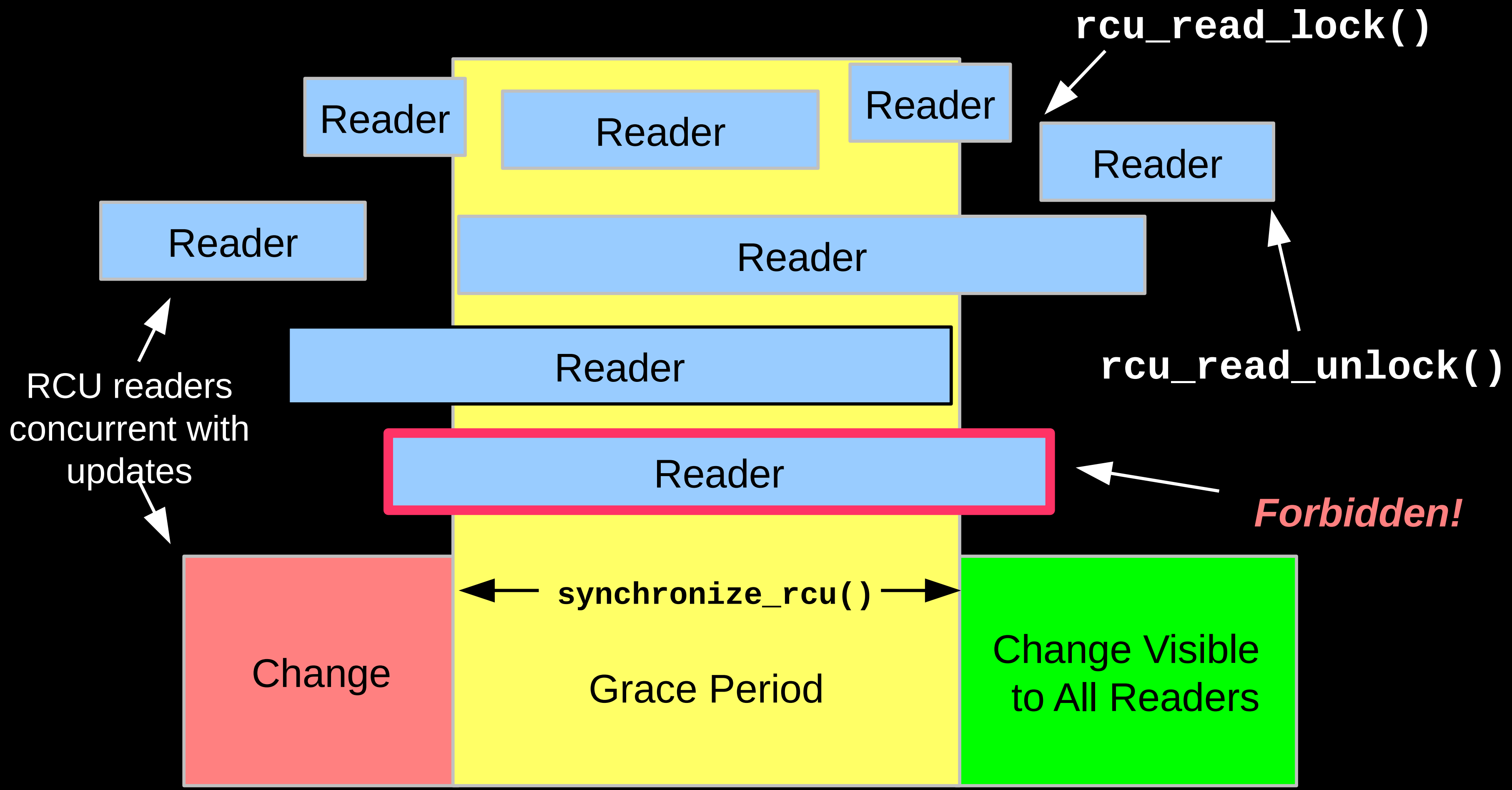


RCU Grace Periods: Conceptual and Graphical Views

RCU Grace Periods: A Conceptual View

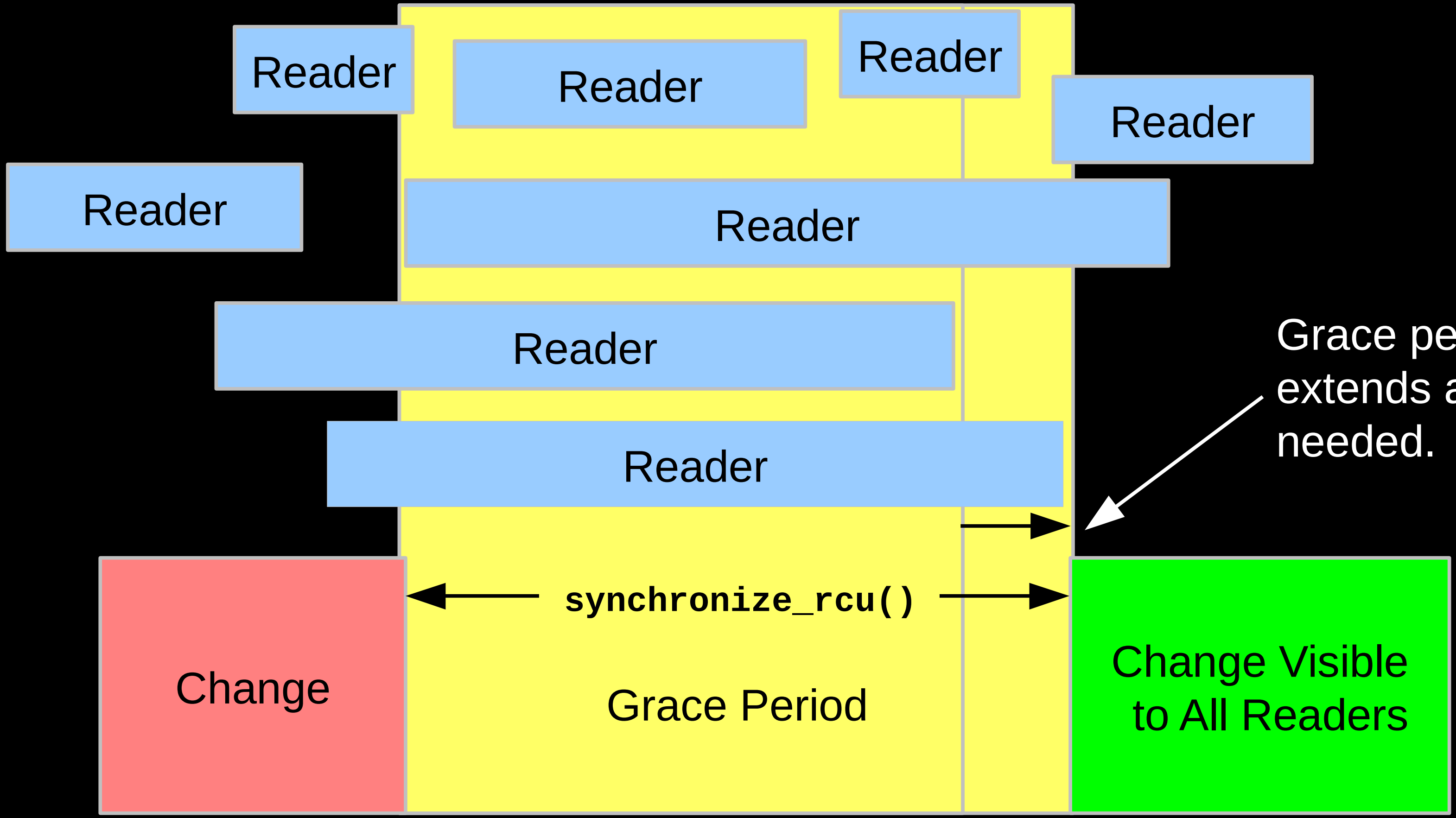
- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread is in at least one quiescent state
  - Ends when all pre-existing readers complete
  - Guaranteed to complete in finite time iff all RCU read-side critical sections are of finite duration
- But what happens if you try to extend an RCU read-side critical section across a grace period?

RCU Grace Periods: A Graphical View



So what happens if you try to extend an RCU read-side critical section across a grace period?

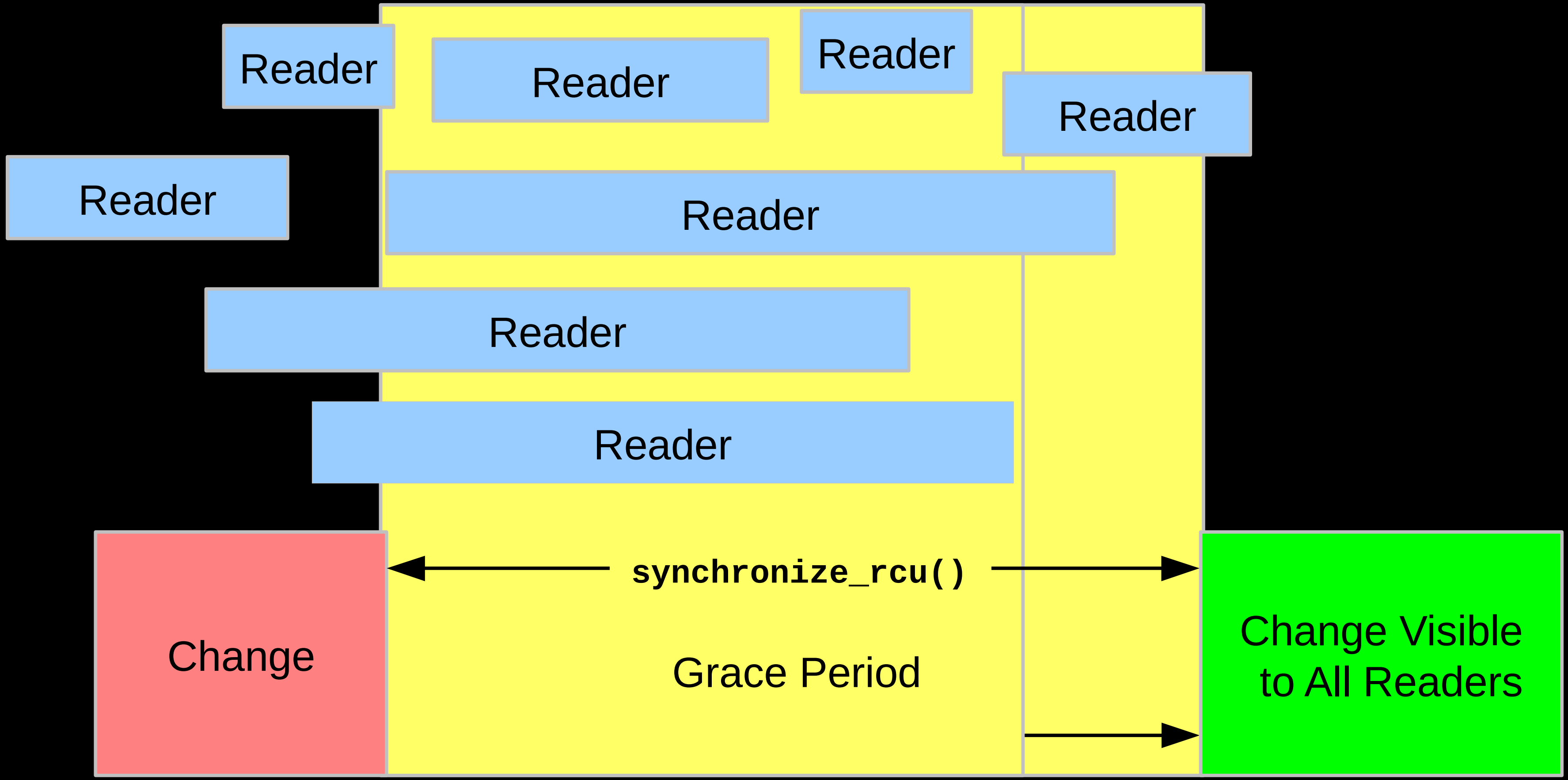
RCU Grace Period: A Self-Repairing Graphical View



Grace period extends as needed.

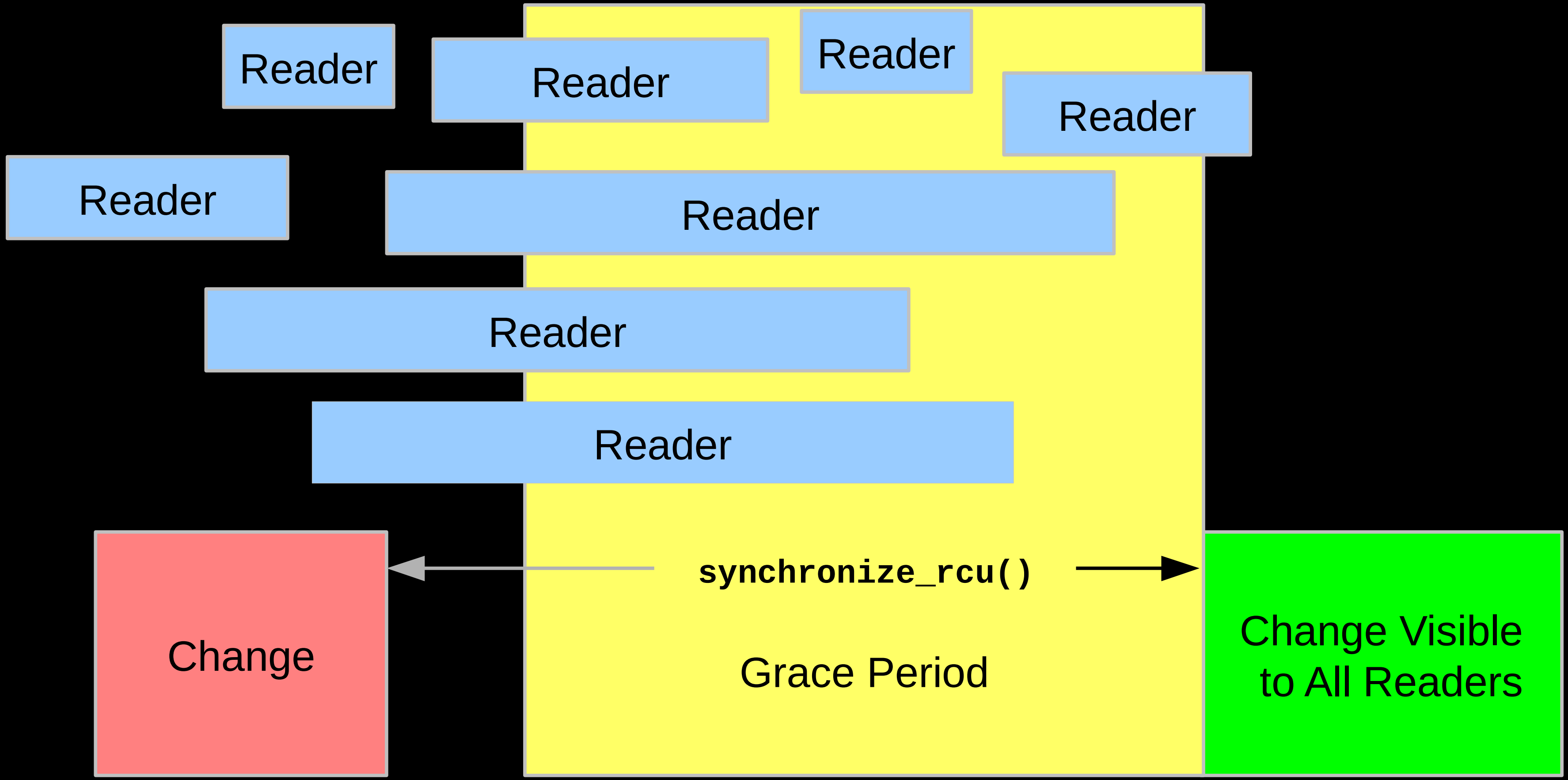
A grace period is not permitted to end until all pre-existing readers have completed.

RCU Grace Period: A Lazy Graphical View



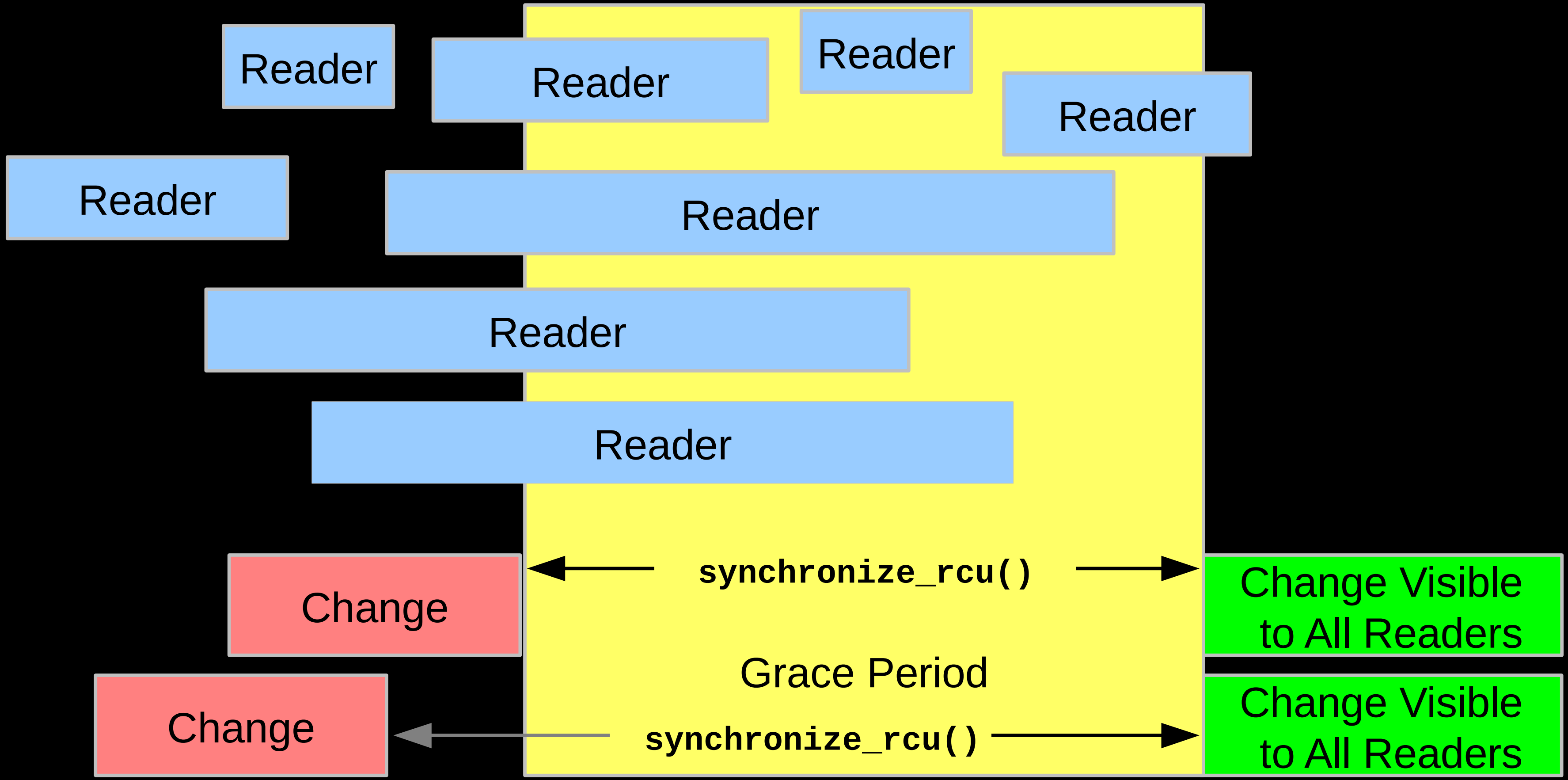
But it is OK for RCU to be lazy and allow a grace period to extend longer than necessary

RCU Grace Period: A *Really* Lazy Graphical View



And it is also OK for RCU to be even more lazy and start a grace period later than necessary
But why is this useful?

RCU Grace Period: A Usefully Lazy Graphical View

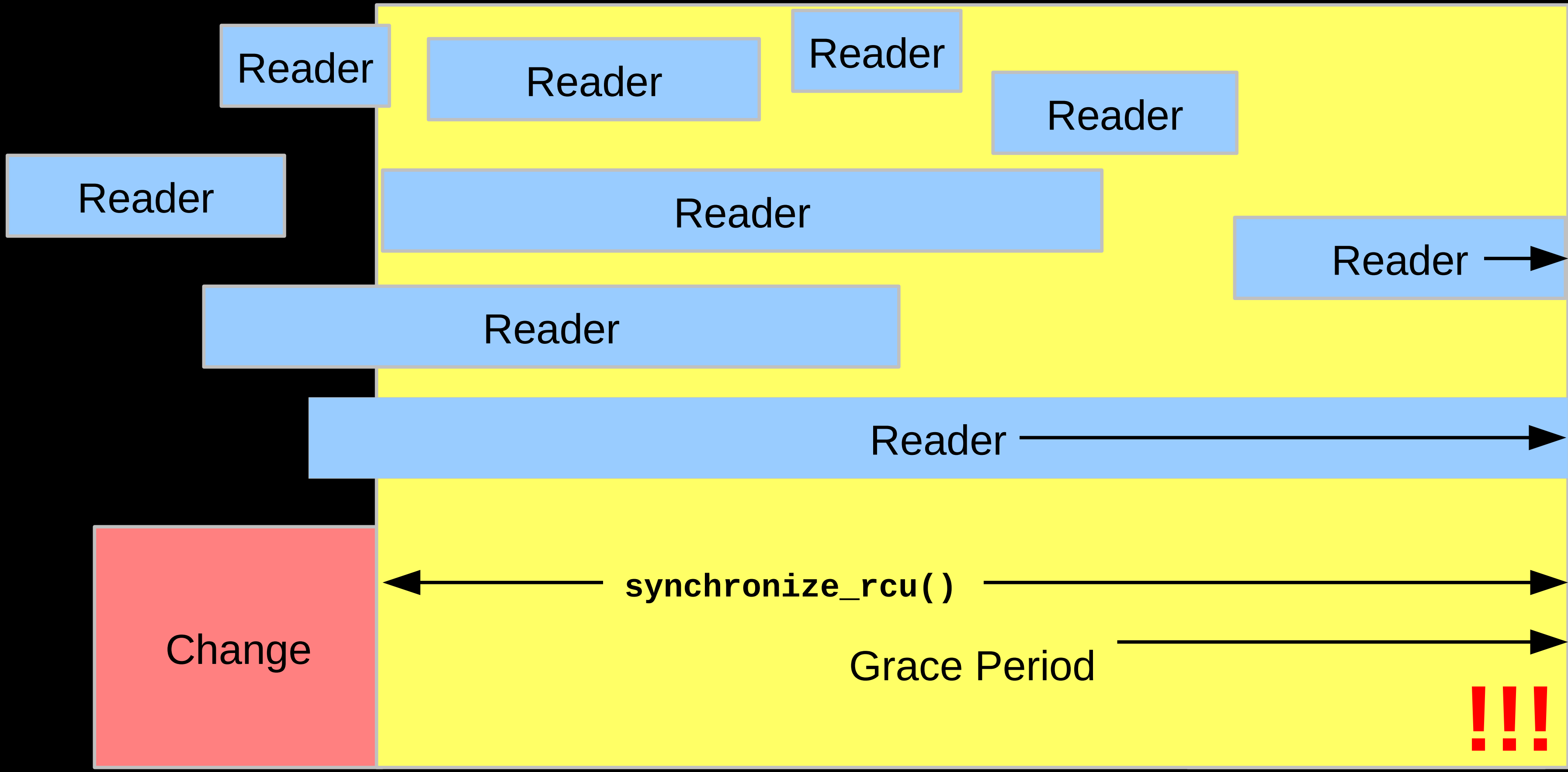


Starting a grace period late can allow it to serve multiple updates, decreasing the per-update RCU overhead. But...

The Costs and Benefits of Laziness

- Starting the grace period later increases the number of updates per grace period, reducing the per-update overhead
- Delaying the end of the grace period increases grace-period latency
- Increasing the number of updates per grace period increases the memory usage
  - Therefore, starting grace periods late is a good tradeoff if memory is cheap and communication is expense, as is the case in modern multicore systems
    - And if real-time threads avoid waiting for grace periods to complete
  - However...

RCU Grace Period: A Too-Lazy Graphical View



And it is OK for the system to complain (or even abort) if a grace period extends too long. Too-long grace periods are likely to result in death by memory exhaustion anyway.

RCU Asynchronous Grace-Period Detection

RCU Asynchronous Grace-Period Detection

- The call_rcu() function registers an RCU callback, which is invoked after a subsequent grace period elapses
- API:

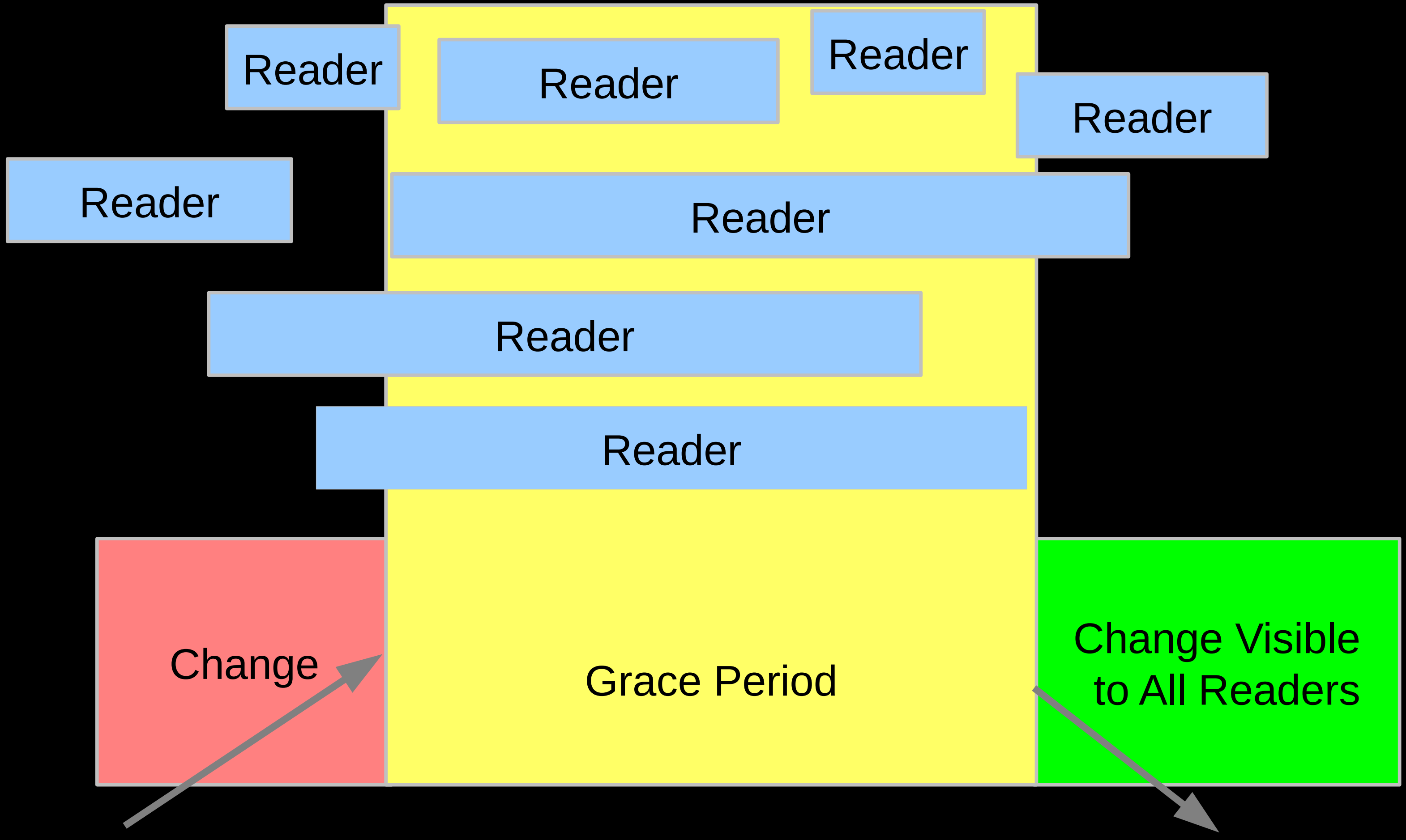
```
call_rcu(struct rcu_head head,
         void (*func)(struct rcu_head *rcu));
```

- The rcu_head structure:

```
struct rcu_head {
        struct rcu_head *next;
        void (*func)(struct rcu_head *rcu);
};
```

- The rcu_head structure is normally embedded within the RCU-protected data structure

RCU Grace Period: An Asynchronous Graphical View

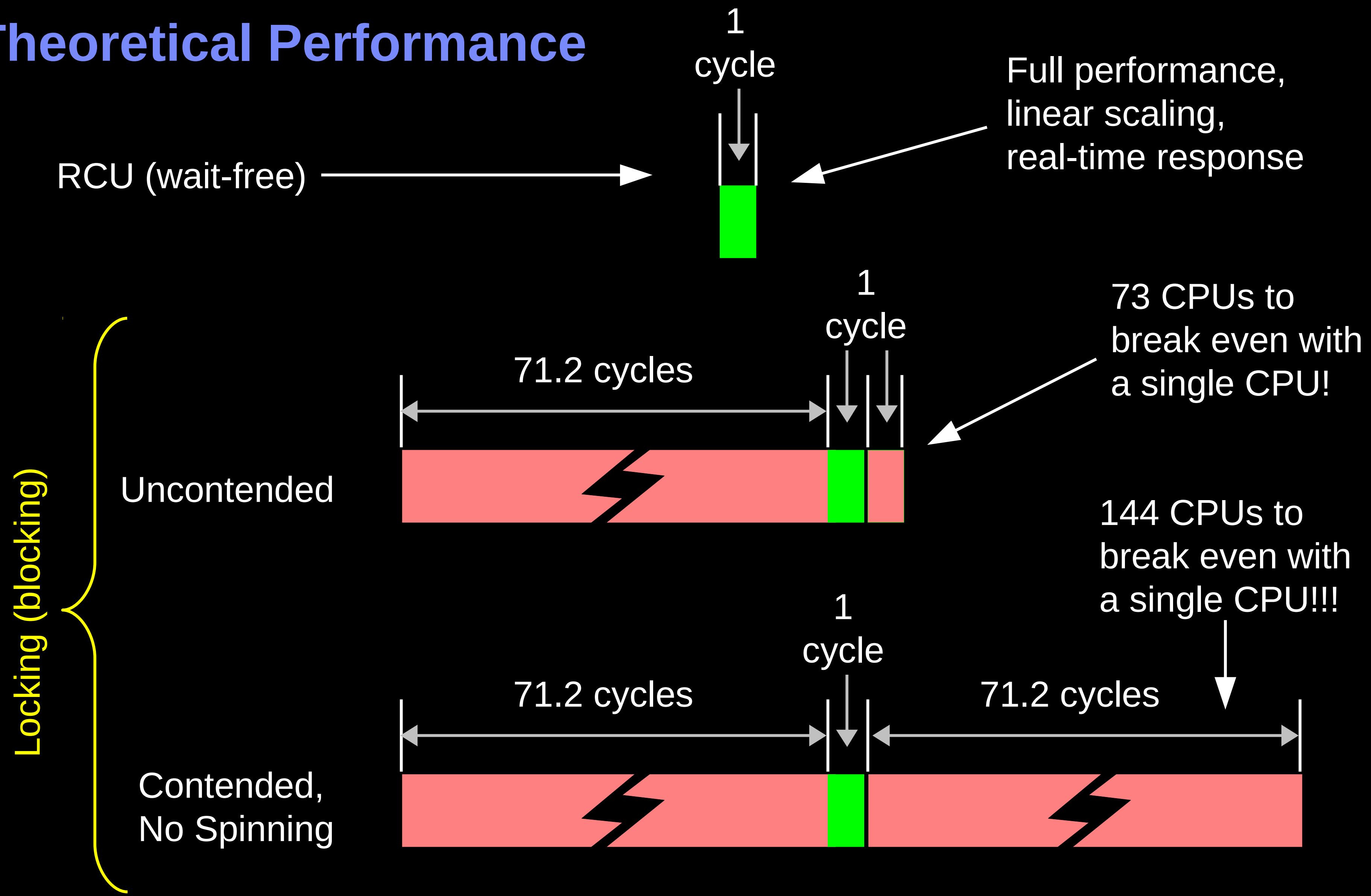


```
call_rcu(&p->rcu, func);
```

```
func(&p->rcu);
```

Performance

Theoretical Performance



RCU (wait-free)

1 cycle

Full performance,
linear scaling,
real-time response

Locking (blocking)

Uncontended

71.2 cycles

1 cycle

73 CPUs to
break even with
a single CPU!

144 CPUs to
break even with
a single CPU!!!

Contended,
No Spinning

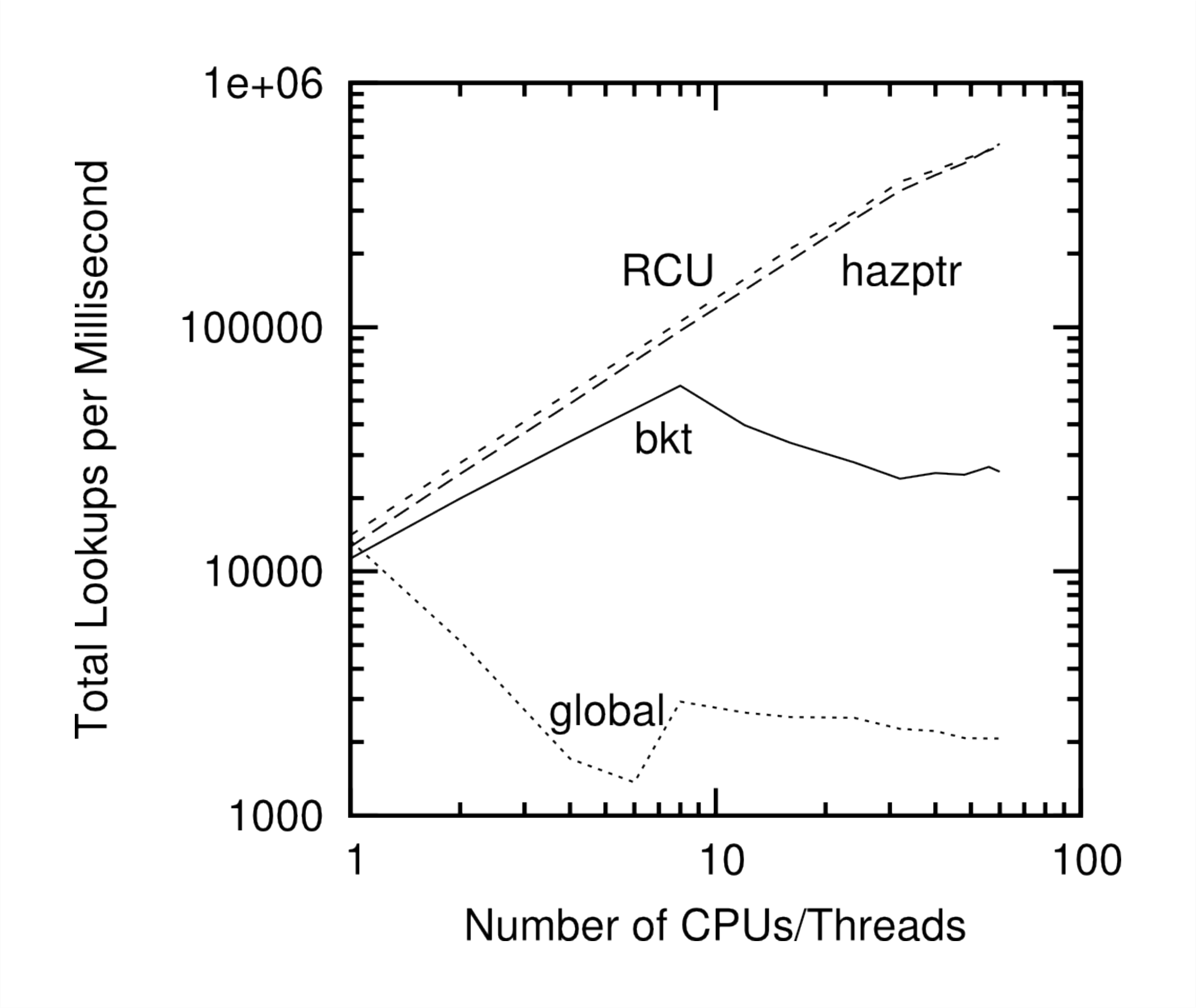
71.2 cycles

1 cycle

71.2 cycles

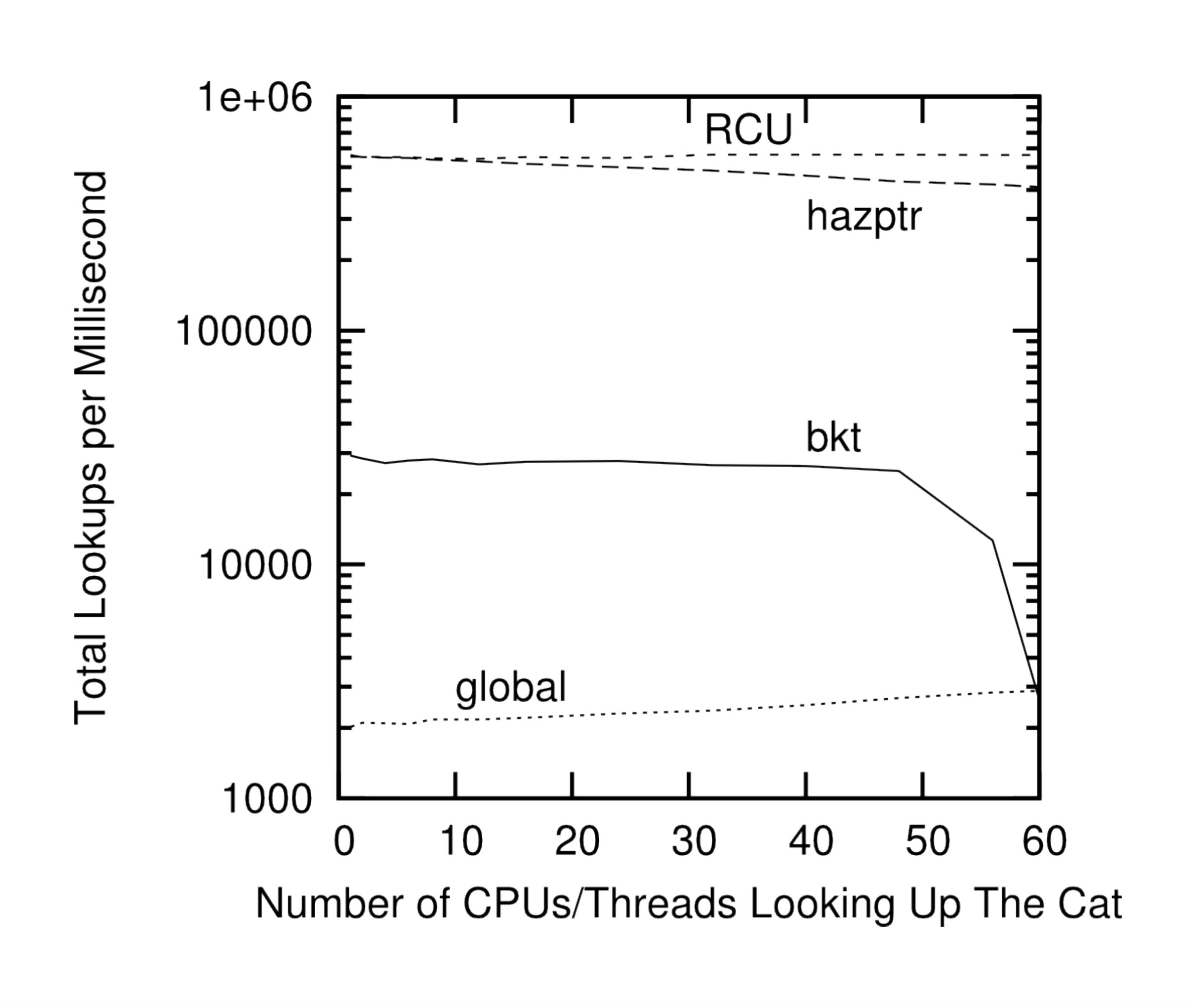
Measured Performance

Schrödinger's Zoo: Read-Only



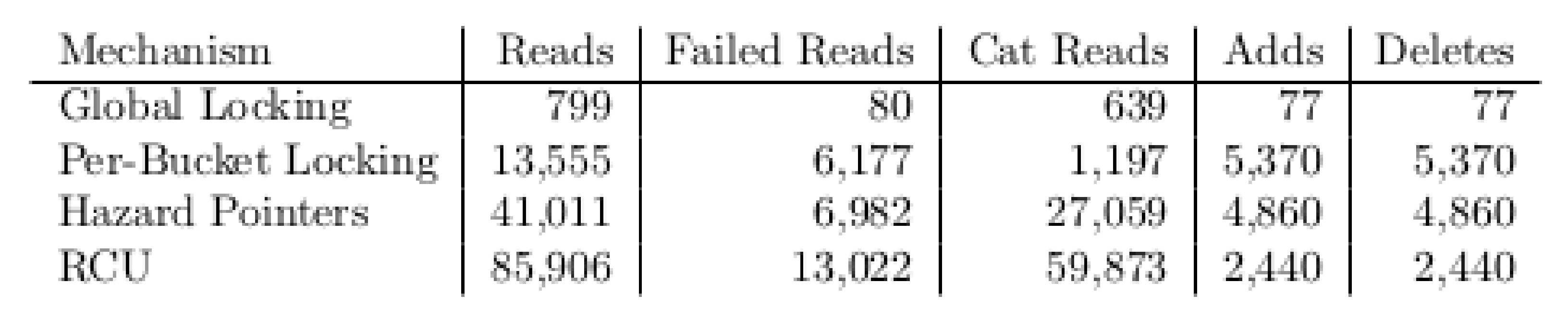
RCU and hazard pointers scale quite well!!!

Schrödinger's Zoo: Read-Only Cat-Heavy Workload



RCU handles locality, hazard pointers not bad, bucket locking horrible!

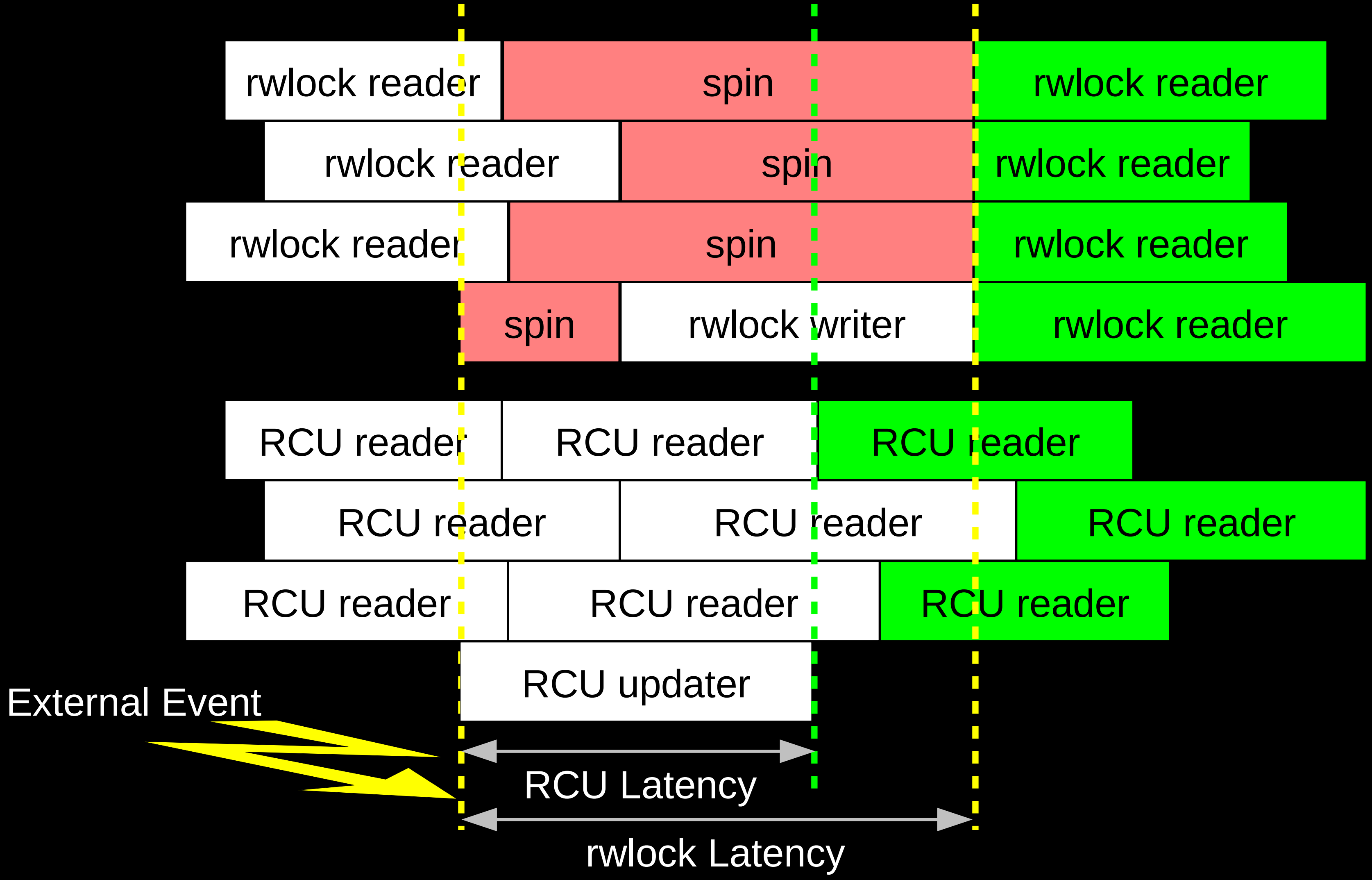
Schrödinger's Zoo: Reads and Updates



Mechanism	Reads	Failed Reads	Cat Reads	Adds	Deletes
Global Locking	799	80	639	77	77
Per-Bucket Locking	13,555	6,177	1,197	5,370	5,370
Hazard Pointers	41,011	6,982	27,059	4,860	4,860
RCU	85,906	13,022	59,873	2,440	2,440

Real-Time Response to Changes

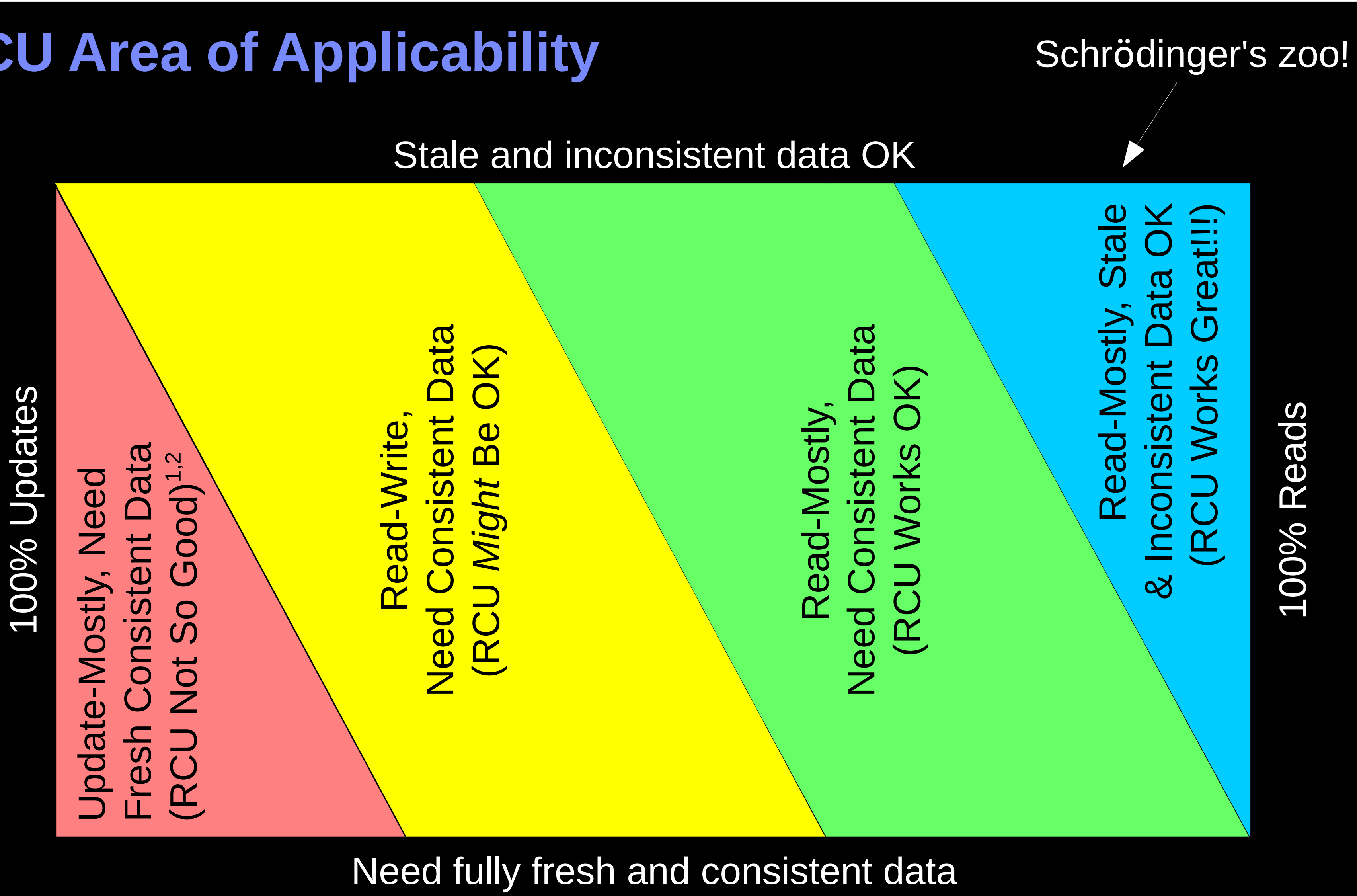
RCU vs. Reader-Writer-Lock Real-Time Latency



RCU Performance: “Free is a *Very* Good Price!!!”

RCU Performance: “Free is a *Very* Good Price!!!” And Nothing Is Faster Than Doing Nothing!!!

RCU Area of Applicability



Schrödinger's zoo!

Stale and inconsistent data OK

100% Updates

Update-Mostly, Need Fresh Consistent Data (RCU Not So Good)[1,2]

Read-Write, Need Consistent Data (RCU *Might* Be OK)

Read-Mostly, Need Consistent Data (RCU Works OK)

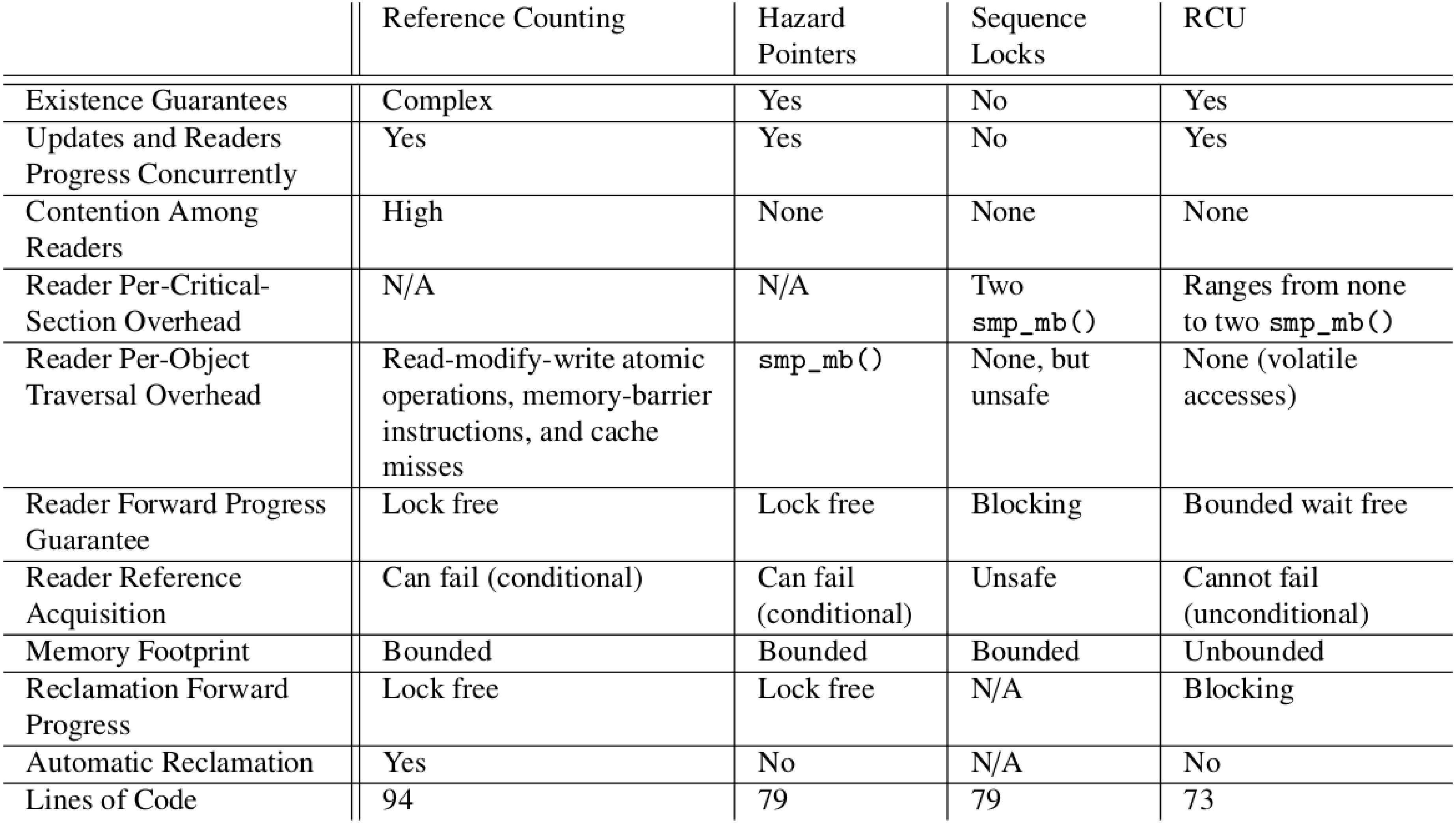
Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!!)

100% Reads

Need fully fresh and consistent data

1. RCU provides ABA protection for update-friendly mechanisms
2. RCU provides bounded wait-free read-side primitives for real-time use

What is RCU?



	Reference Counting	Hazard Pointers	Sequence Locks	RCU
Existence Guarantees	Complex	Yes	No	Yes
Updates and Readers Progress Concurrently	Yes	Yes	No	Yes
Contention Among Readers	High	None	None	None
Reader Per-Critical-Section Overhead	N/A	N/A	Two `smp_mb()`	Ranges from none to two `smp_mb()`
Reader Per-Object Traversal Overhead	Read-modify-write atomic operations, memory-barrier instructions, and cache misses	`smp_mb()`	None, but unsafe	None (volatile accesses)
Reader Forward Progress Guarantee	Lock free	Lock free	Blocking	Bounded wait free
Reader Reference Acquisition	Can fail (conditional)	Can fail (conditional)	Unsafe	Cannot fail (unconditional)
Memory Footprint	Bounded	Bounded	Bounded	Unbounded
Reclamation Forward Progress	Lock free	Lock free	N/A	Blocking
Automatic Reclamation	Yes	No	N/A	No
Lines of Code	94	79	79	73

Table 9.5: Which Deferred Technique to Choose?

Existence Guarantees

- Purpose: Avoid data being yanked from under reader
- Reference counting (also non-blocking synchronization)
  - Possible, but complex and error-prone
- Hazard pointers: Yes
- Sequence locks: No
  - You just get told later that something might have been yanked
- RCU: Yes

Reader/Writer Concurrent Forward Progress

- Purpose: Avoid starvation independent of workload
- Reference counting: Yes
- Hazard pointers; Yes
- Sequence locks: No, updates roll back readers
- RCU: Yes

Avoid Read-Side Contention

- Purpose: Scalability, performance, forward progress
- Reference counting: No, high memory contention
- Hazard pointers: Yes
- Sequence locking: Yes
- RCU: Yes

Degree of Read-Side Critical-Section Overhead

- Purpose: Low overhead means faster execution
- Reference counting: None (no critical sections)
- Hazard pointers: None (no critical sections)
- Sequence locks: Two full memory barriers
- RCU:
  - Ranges from none (QSBR) to two full memory barriers (SRCU)

Read-Side Per-Object Traversal Overhead

- Purpose: Low overhead for faster execution
- Reference counting: RMW atomic operations, memory-barrier instructions, and cache misses
- Hazard pointers: smp_mb(), but can eliminate with operating-system membarrier support
- Sequence locking: Kernel panic!!!
- RCU: None (except on DEC Alpha)

Read-Side Forward Progress Guarantee

- Purpose: Meet response-time commitments
- Reference counting: Lock free
- Hazard pointers: Lock free
- Sequence locks: Blocking (can wait on updater)
- RCU: Population-oblivious bounded wait-free

Read-Side Reference Acquisition

- Purpose: Must client code retry read-side traversals?
- Reference counting: Traverals can fail, requiring retry
- Hazard pointers: Traverals can fail, requiring retry
- Sequence locking: Kernel can panic
- RCU: Traverals guaranteed to succeed, no retry needed

Memory Footprint

- Purpose: Small memory footprints are good!
  - Especially if you are as old as I am!!!
- Reference counting: Bounded (number of active references)
- Hazard pointers: Bounded (number of active references, though tight bound incurs CPU overhead)
- Sequence locks: Bounded (especially given unsafe traversal)
- RCU: Unbounded or updaters delayed

Reclamation Forward Progress

- Purpose: Tight memory footprint independent of workload
- Reference counting: Lock free
- Hazard pointers: Lock free
- Sequence locking: N/A
- RCU: Blocking: Single reader can block reclamation

Automatic Reclamation

- Purpose: Simplify memory management
- Reference counting: Yes
- Hazard pointers: No, but working on it
- Sequence locking: N/A
- RCU: No, but working on it

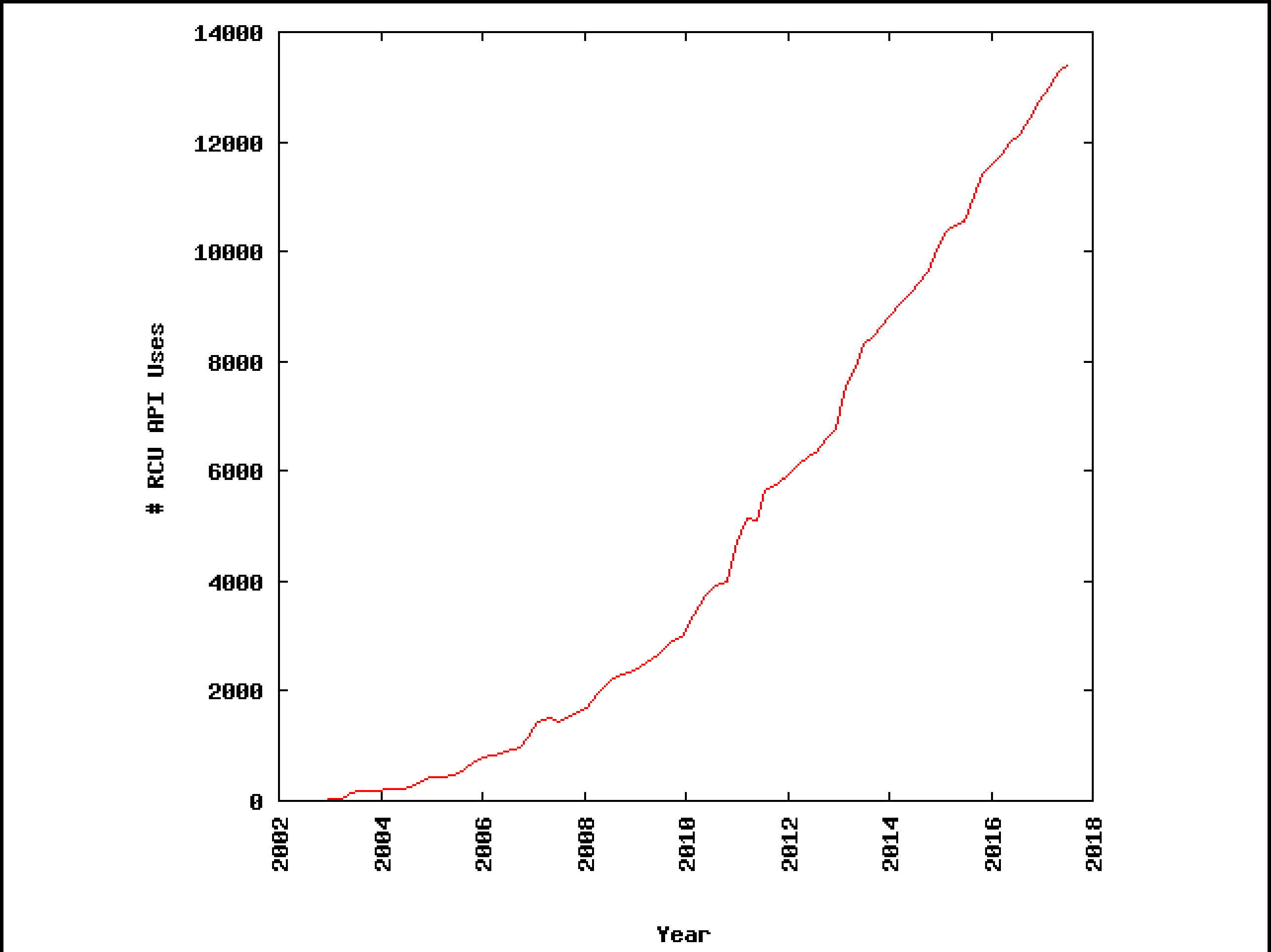
Lines of Code for Pre-BSD Routing Table

- Reference counting: 94 (but buggy)
- Hazard pointers: 79
- Sequence locks: 79 (but buggy)
- RCU: 73

Different Design Points!

- Locking is still the workhorse for production software
- Non-blocking synchronization where it works well
- Reference counting OK on small systems or for rarely accessed portions of larger systems, and provide tight bounts on memory. Traversals subject to retry.
- Hazard pointers handle large systems, provide tight bounds on memory, excellent scalability, and decent traversal performance. Traversals subject to retry.
- Sequence locks need one of the other approaches
- RCU handles huge systems, excellent scalability and traversal overhead, no-retry traversals. Unbounded memory footprint.

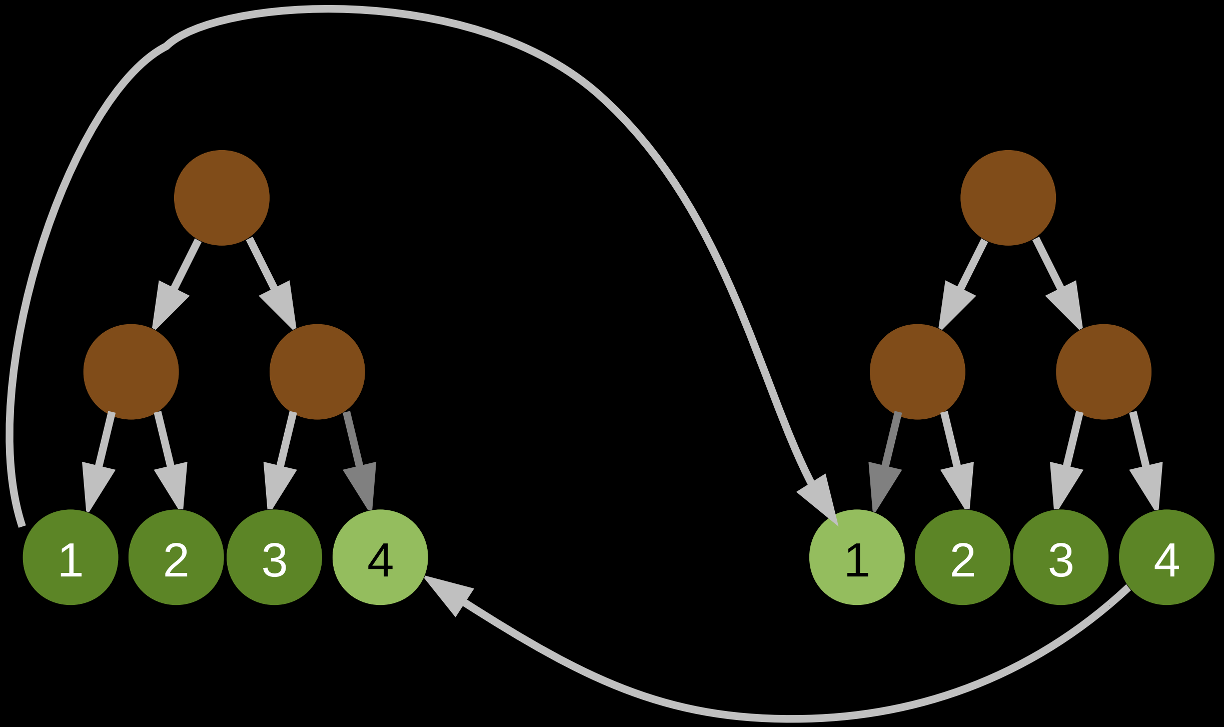
RCU Applicability to the Linux Kernel



Complex Atomic-To-Reader Updates, Take 2

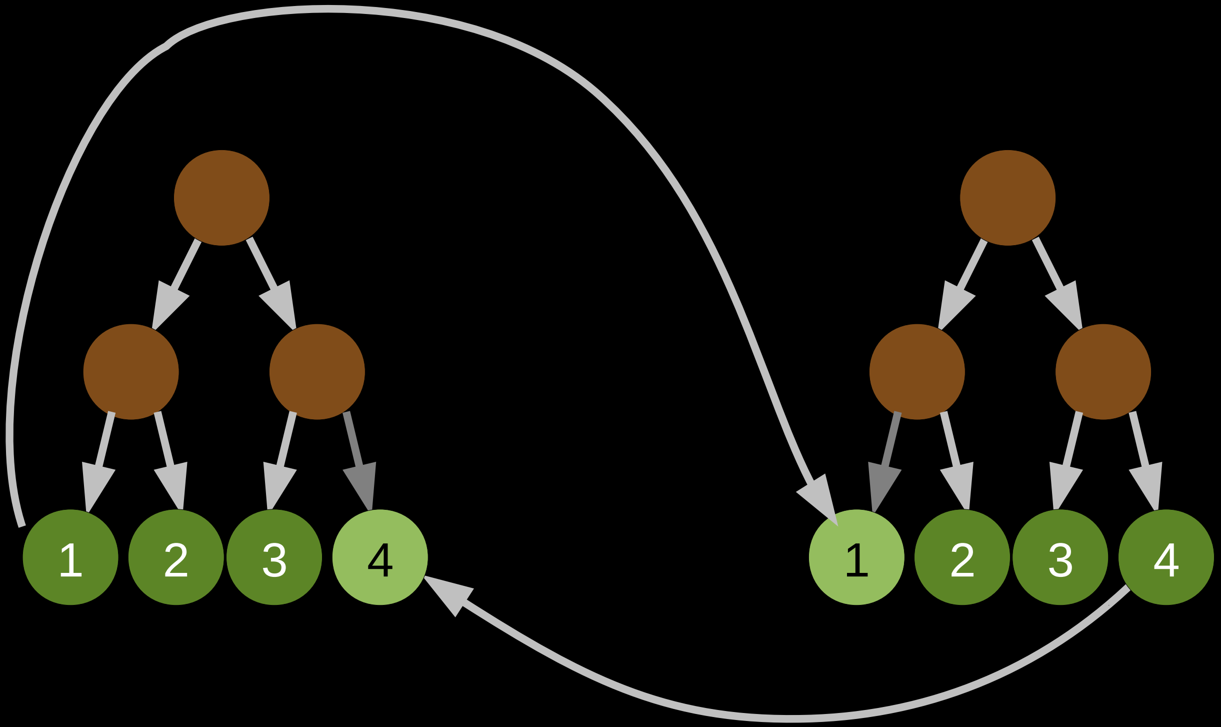
Complex Atomic-To-Reader Updates, Take 2
Atomic Multi-Structure Update: Issaquah Challenge

Atomic Multi-Structure Update: Issaquah Challenge



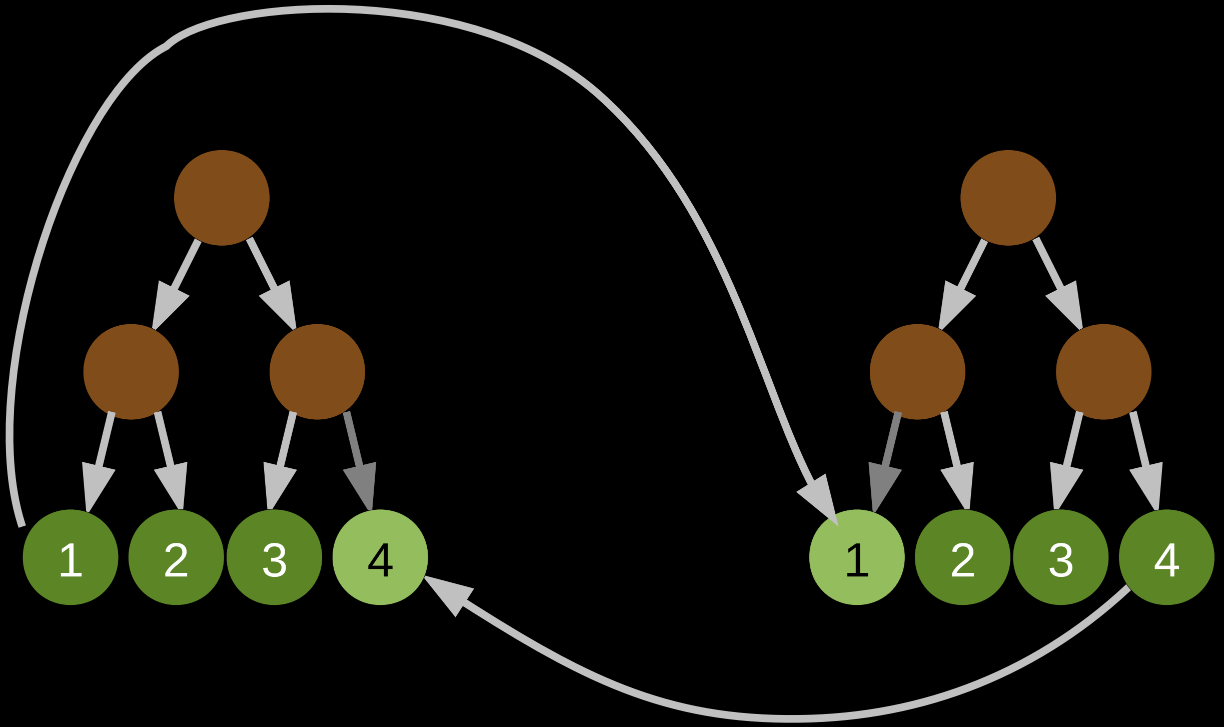
Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree

Atomic Multi-Structure Update: Issaquah Challenge



Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!

Atomic Multi-Structure Update: Issaquah Challenge



Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!
Hence, most locking solutions "need not apply"

Recall Applicable Laws of Physics...

- The finite speed of light
- The atomic nature of matter

- We therefore avoid *unnecessary* updates!!!

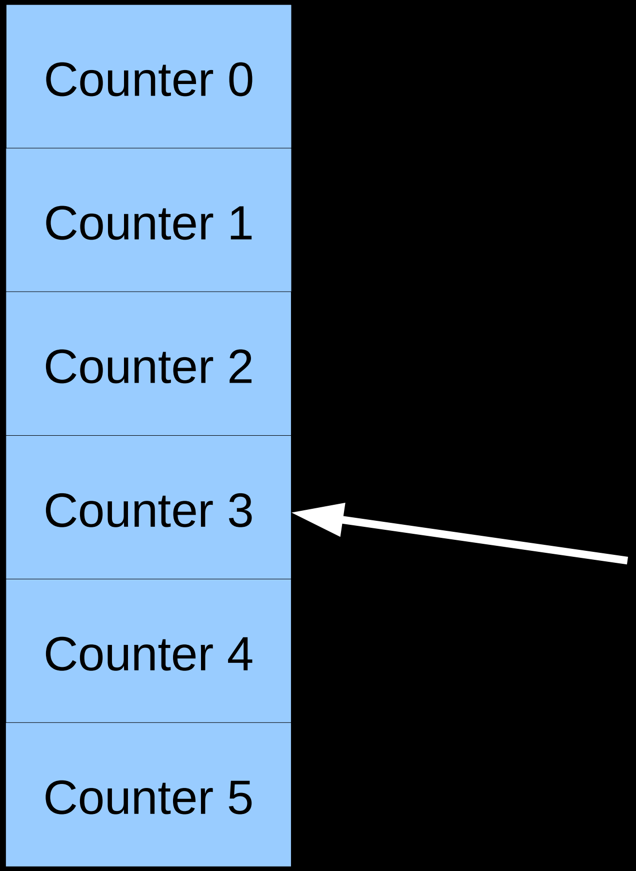
Update-Heavy Workloads Painful for Parallelism!!! But There Are Some Special Cases...

But There Are Some Special Cases

- Per-CPU/thread processing (perfect partitioning)
  - Huge number of examples, including the per-thread/CPU stack
  - We will look at split counters
- Read-only traversal to location being updated
  - Key to solving the Issaquah Challenge
- Trivial Lock-Based Concurrent Deque???

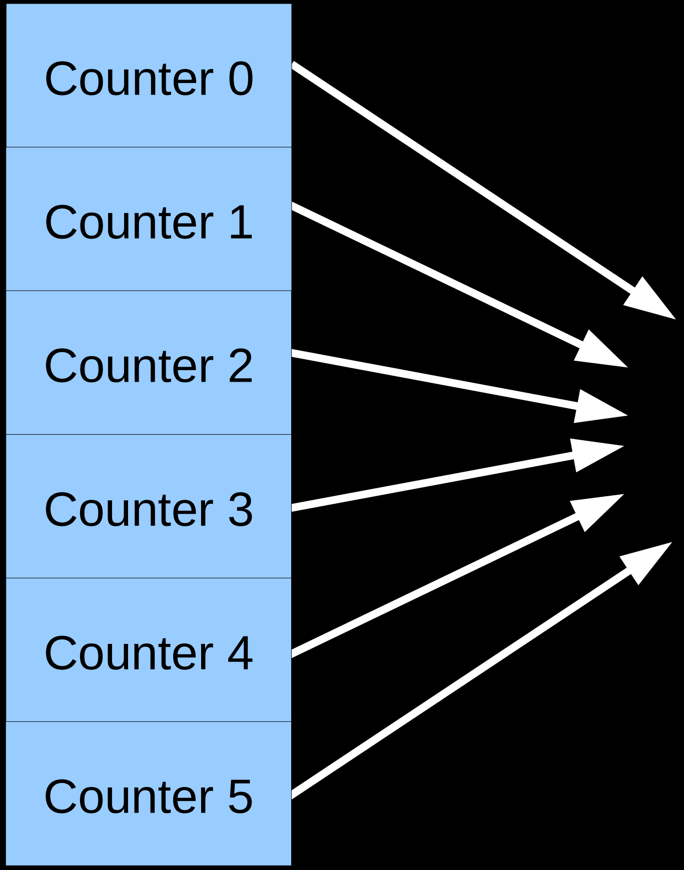
Split Counters

Split Counters Diagram



Increment only your own counter

Split Counters Diagram



Sum all counters

While they continue changing

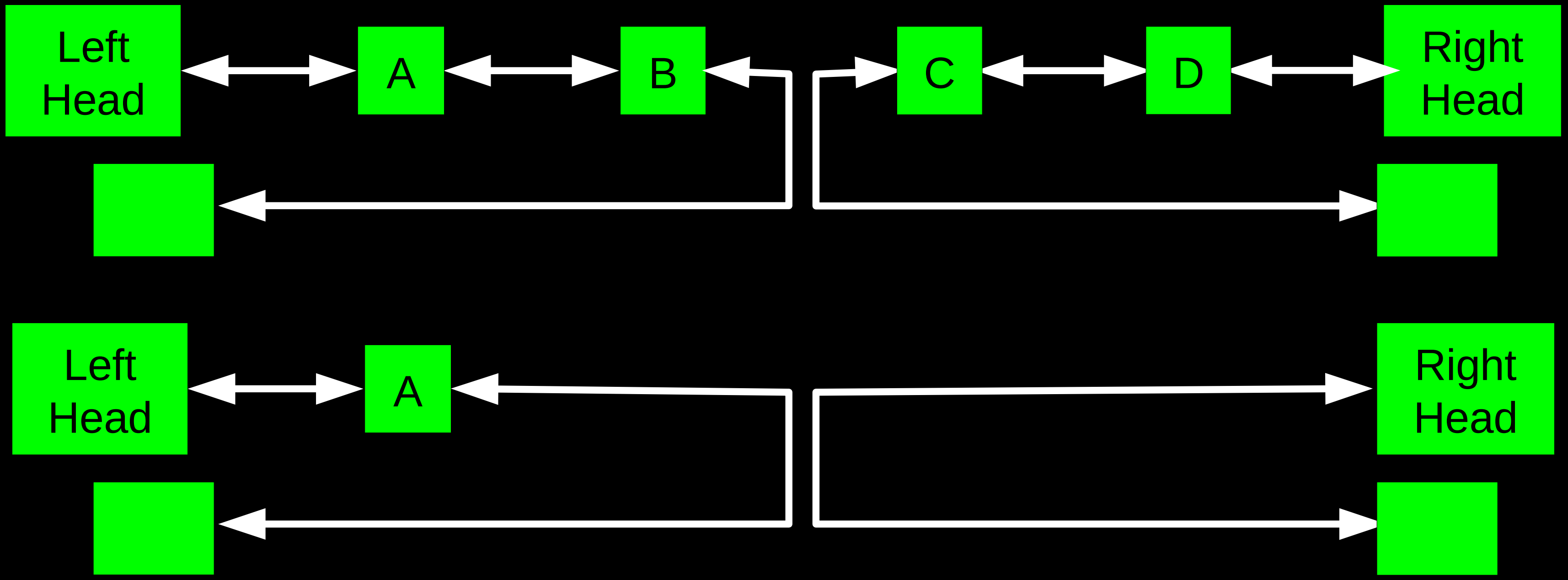
Split Counters Lesson

- Updates need not slow us down – if we maintain good locality
- For the split counters example, in the common case, each thread only updates its own counter
  - Reads of all counters should be rare
  - If they are not rare, use some other counting algorithm
  - There are a lot of them, see “Counting” chapter of “Is Parallel Programming Hard, And, If So, What Can You Do About It?” (http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html)

Trivial Lock-Based Concurrent Dequeue

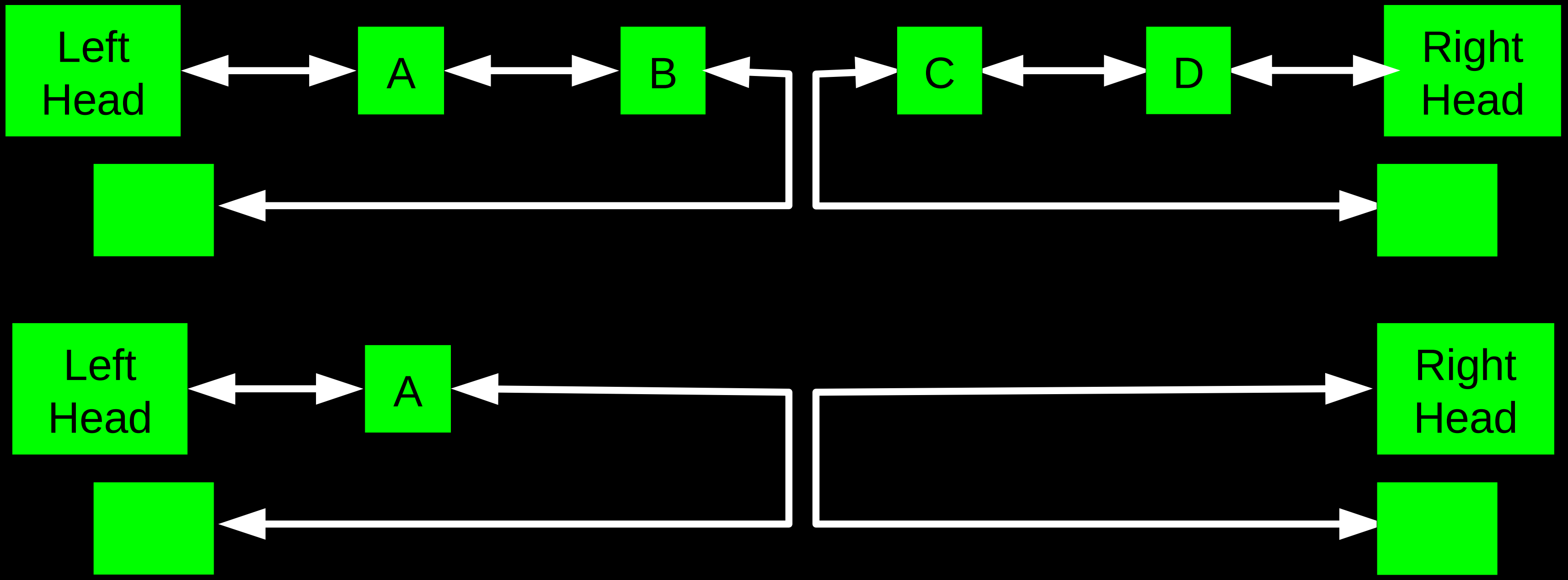
Trivial Lock-Based Concurrent Dequeue

- Use two lock-based dequeues
  - Can always insert concurrently: grab dequeue's lock
  - Can always remove concurrently unless one or both are empty
    - If yours is empty, grab both locks in order!



Trivial Lock-Based Concurrent Dequeue

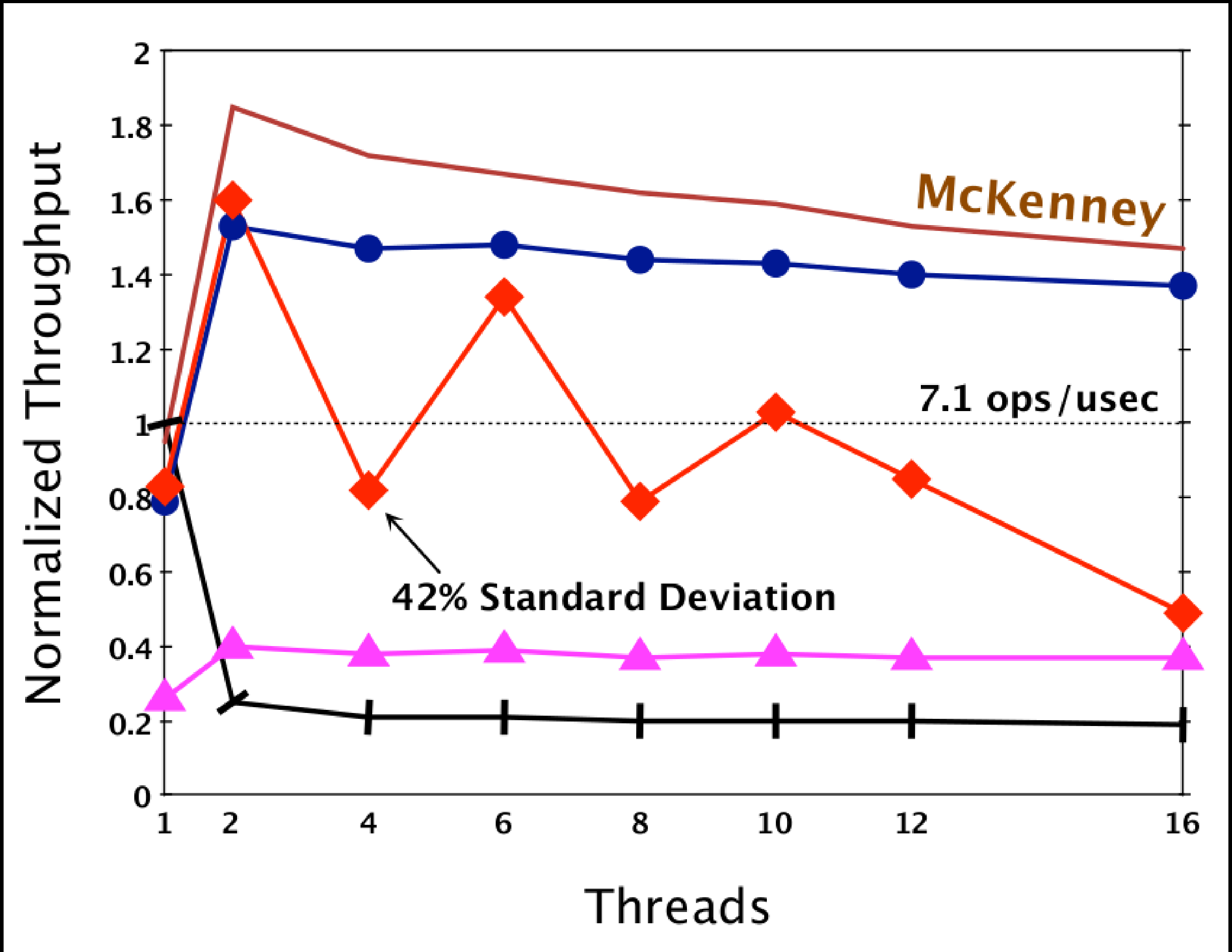
- Use two lock-based dequeues
  - Can always insert concurrently: grab dequeue's lock
  - Can always remove concurrently unless one or both are empty
    - If yours is empty, grab both locks in order!
- But why push all your data through one dequeue???



Trivial Lock-Based Concurrent Dequeue Performance

- Dalessandro et al., “Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory”, ASPLOS'11, March 5-11, Newport Beach, California, USA
  - See "Deque benchmark" subsection of section 4.2 on page 6, especially Figure 7a (next slide)
  - Lock-based dequeue beats all STM algorithms

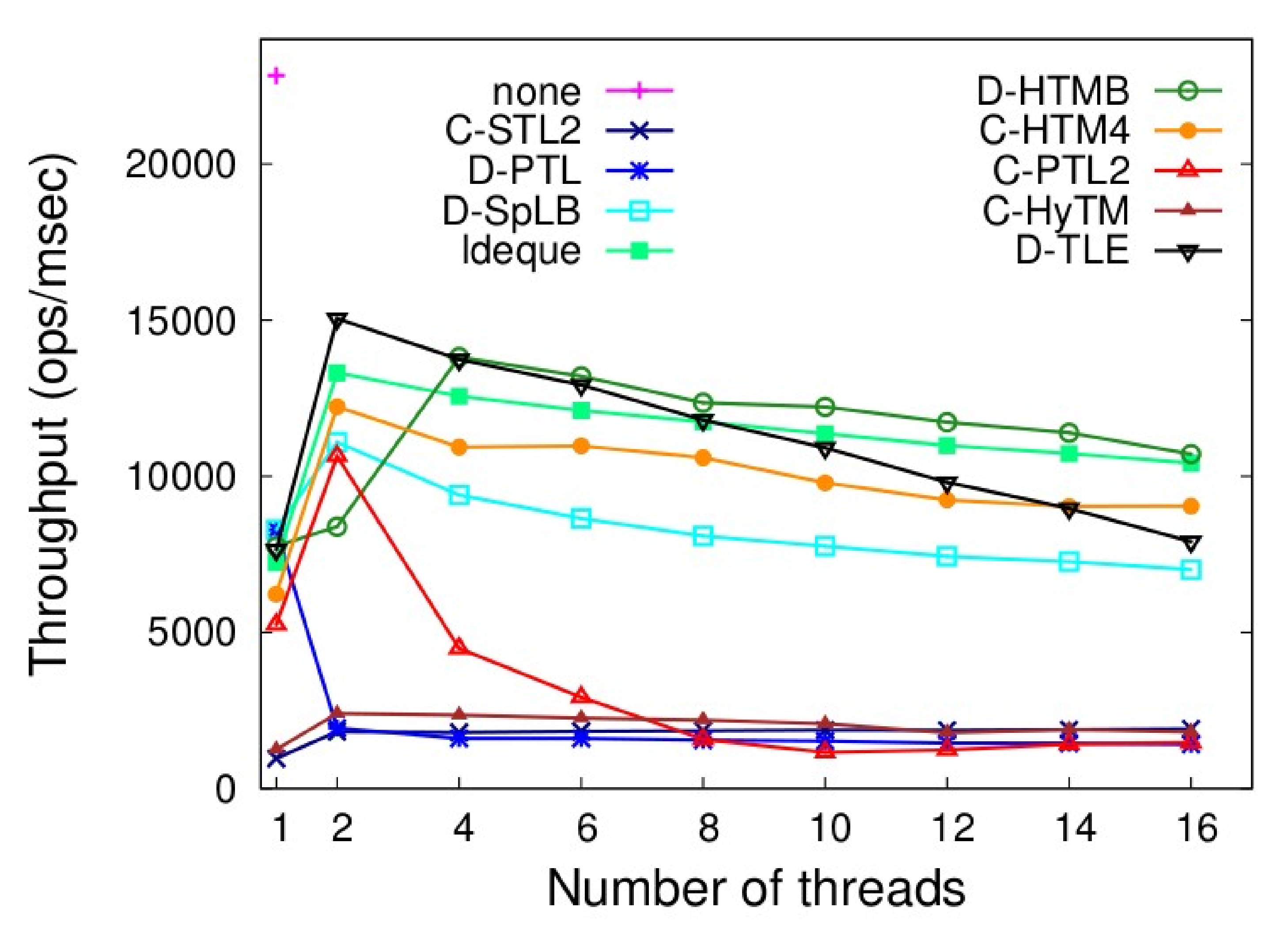
Dalessandro et al. Figure 7a:



Trivial Lock-Based Concurrent Dequeue Performance

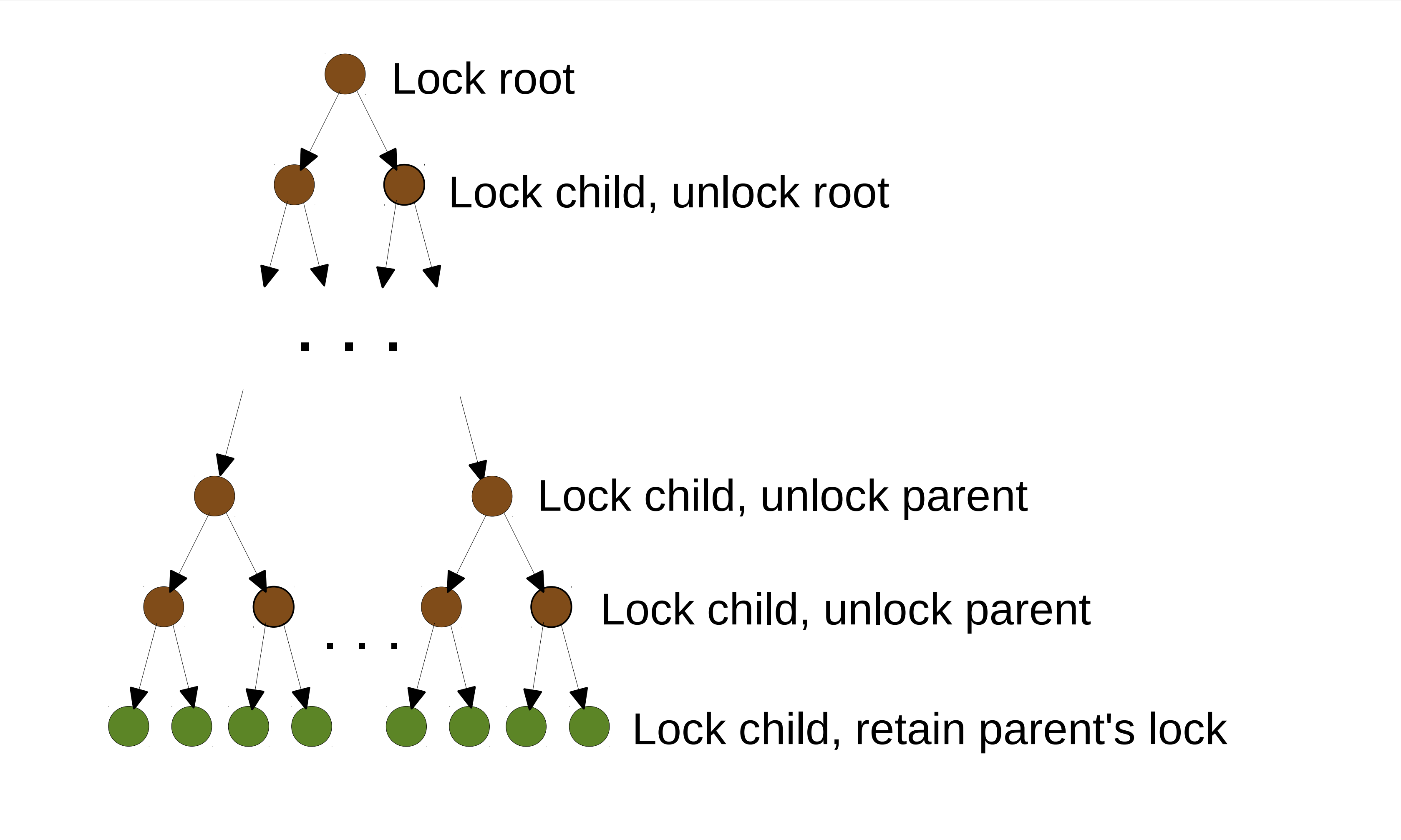
- Dice et al., “Simplifying concurrent algorithms by exploiting hardware transactional memory”, SPAA'10, June 13-15, 2010, Thira, Santorini, Greece.
  - See Figure 1 and discussion in Section 3 on page 2
  - Lock-based dequeue beats all HTM algorithms at some point
- Both sets of authors were exceedingly gracious, without the need for a Code of Conflict

Dice et al., Figure 1



Read-Only Traversal To Location Being Updated

Why Read-Only Traversal To Update Location?


Lock root
Lock child, unlock root
. . .
Lock child, unlock parent
. . .
Lock child, unlock parent
Lock child, retain parent's lock

Lock contention despite read-only accesses!

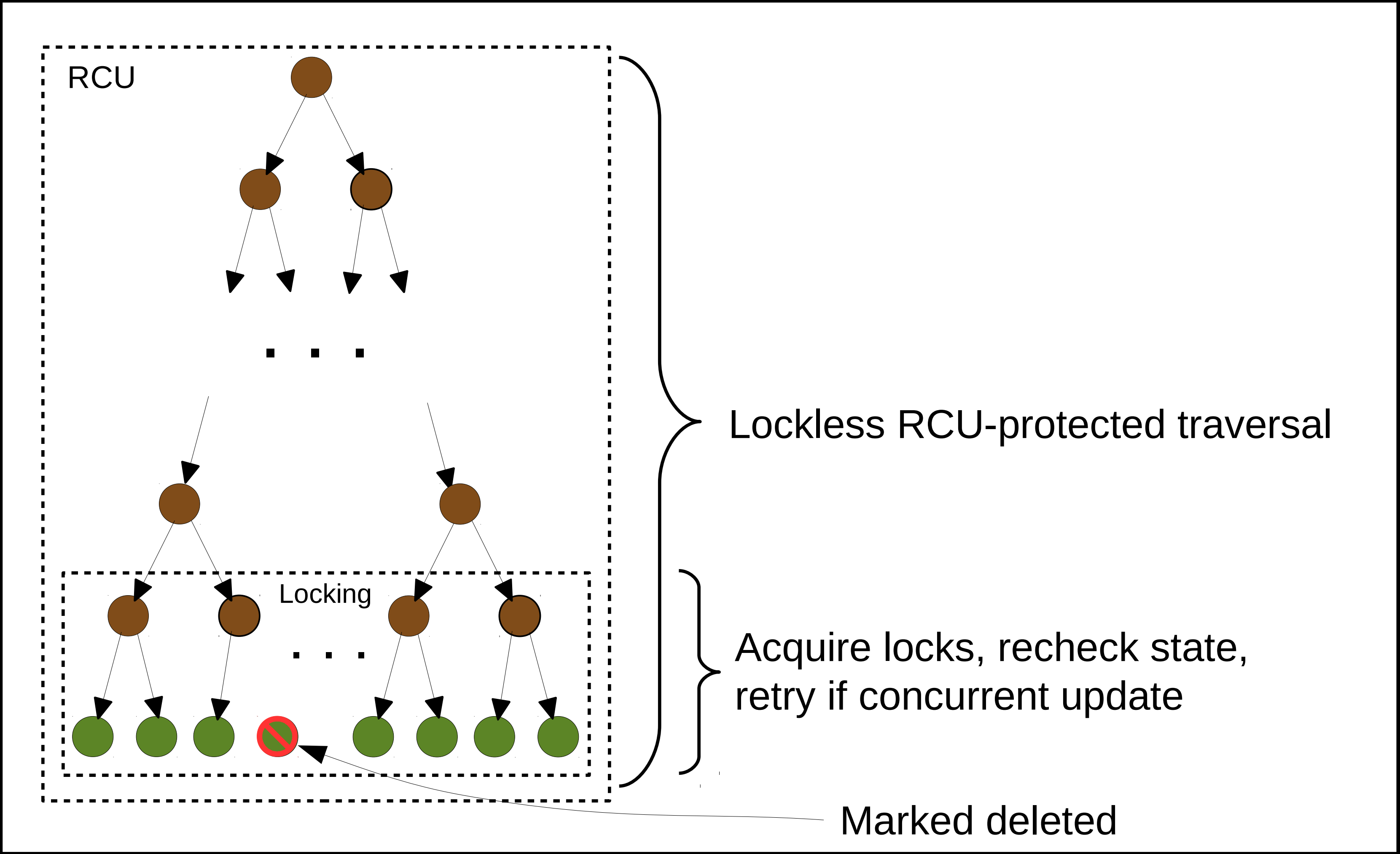
And This Is Another Reason Why We Have RCU!

- (You can also use garbage collectors, hazard pointers, reference counters, etc.)
- Design principle: Avoid expensive operations in read-side code
- As noted earlier, lightest-weight conceivable read-side primitives

  ```
  /* Assume non-preemptible (run-to-block) environment. */
  #define rcu_read_lock()
  #define rcu_read_unlock()
  ```

Better Read-Only Traversal To Update Location

Deletion-Flagged Read-Only Traversal

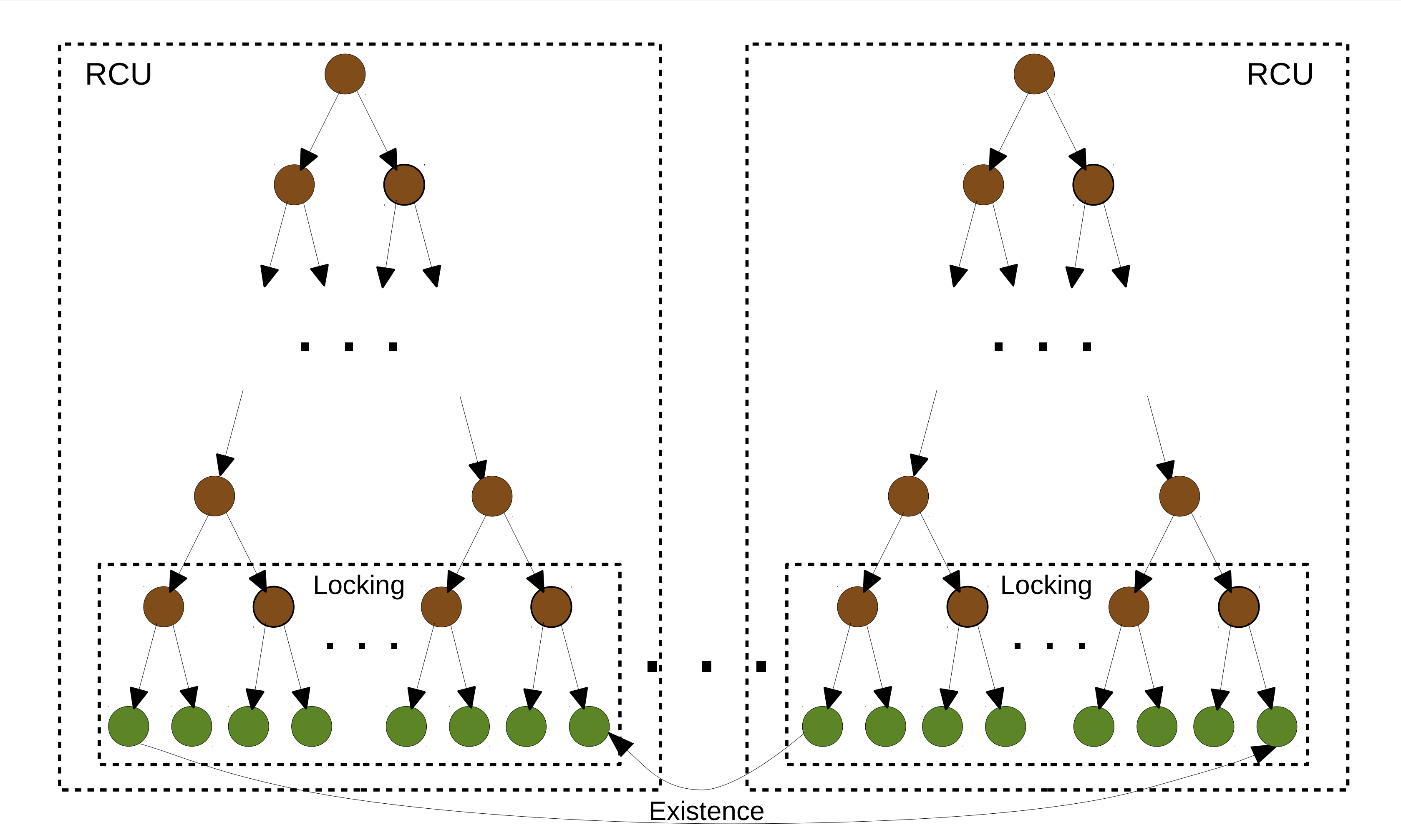

RCU
Locking
. . .
. . .
Lockless RCU-protected traversal
Acquire locks, recheck state,
retry if concurrent update
Marked deleted

Read-Only Traversal To Location Being Updated

- Focus contention on portion of structure being updated
  - And preserve locality of reference to different parts of structure
- Of course, full partitioning is better!
- Read-only traversal technique citations:
  - Arbel & Attiya, “Concurrent Updates with RCU: Search Tree as an Example”, PODC'14 (very similar lookup, insert, and delete)
  - McKenney, Sarma, & Soni, “Scaling dcache with RCU”, Linux Journal, January 2004
  - And possibly: Pugh, “Concurrent Maintenance of Skip Lists”, University of Maryland Technical Report CS-TR-2222.1, June 1990
  - And maybe also: Kung & Lehman, “Concurrent Manipulation of Binary Search Trees”, ACM TODS, September, 1980

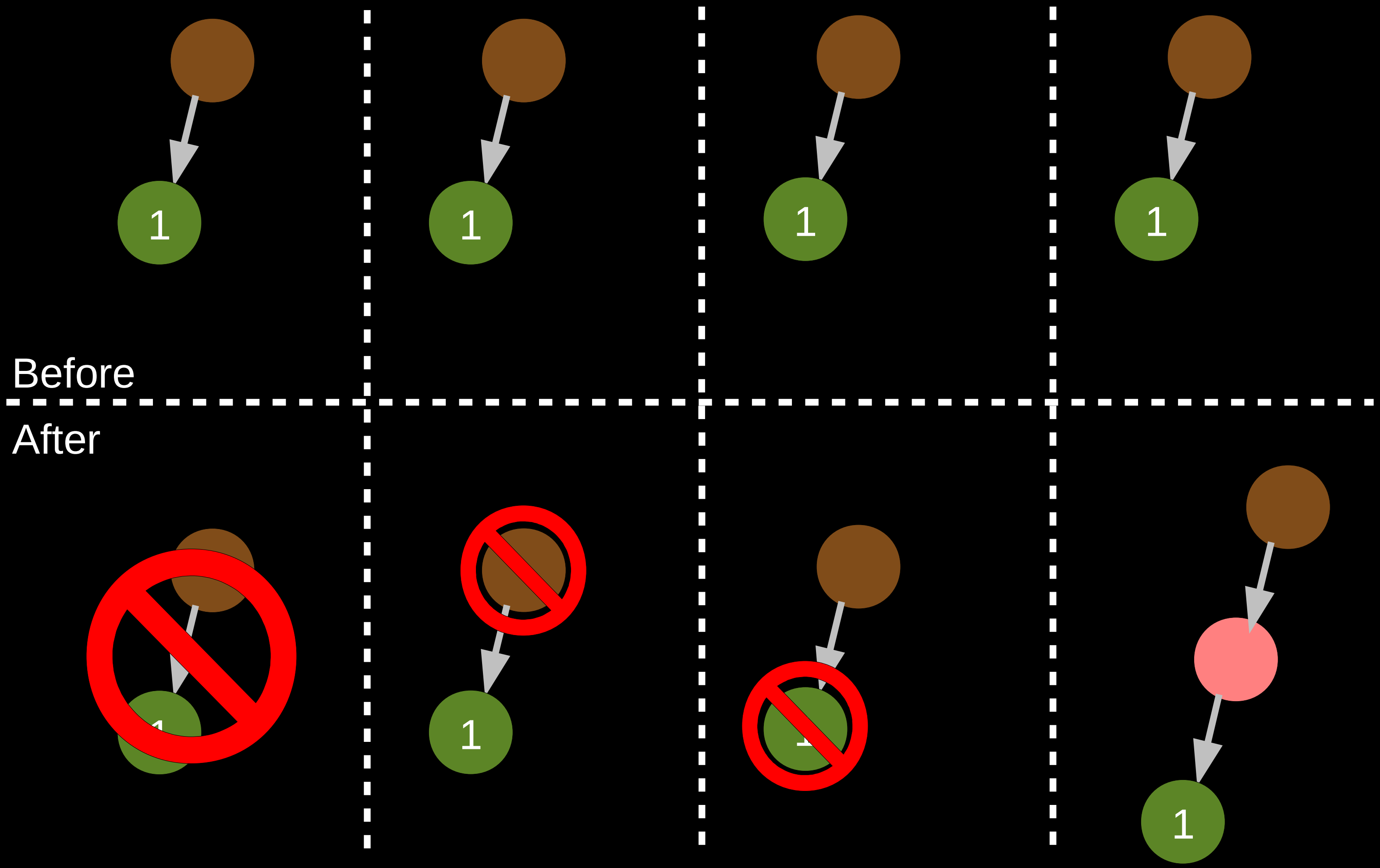
Issaquah Challenge: One Solution

Locking Regions for Binary Search Tree



In many cases, can implement existence as simple wrapper!

Possible Upsets While Acquiring Locks...



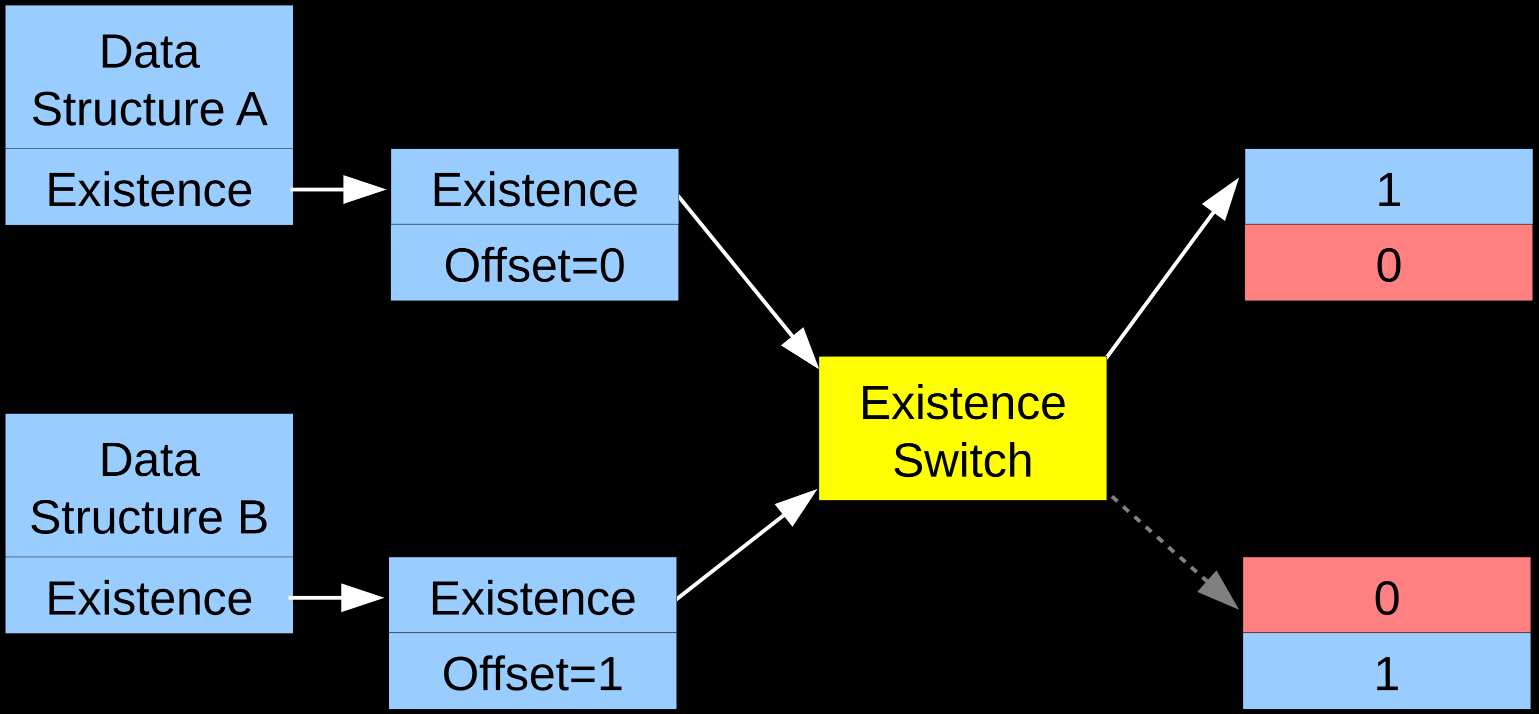
What to do?
Drop locks and retry!!!

Existence Structures

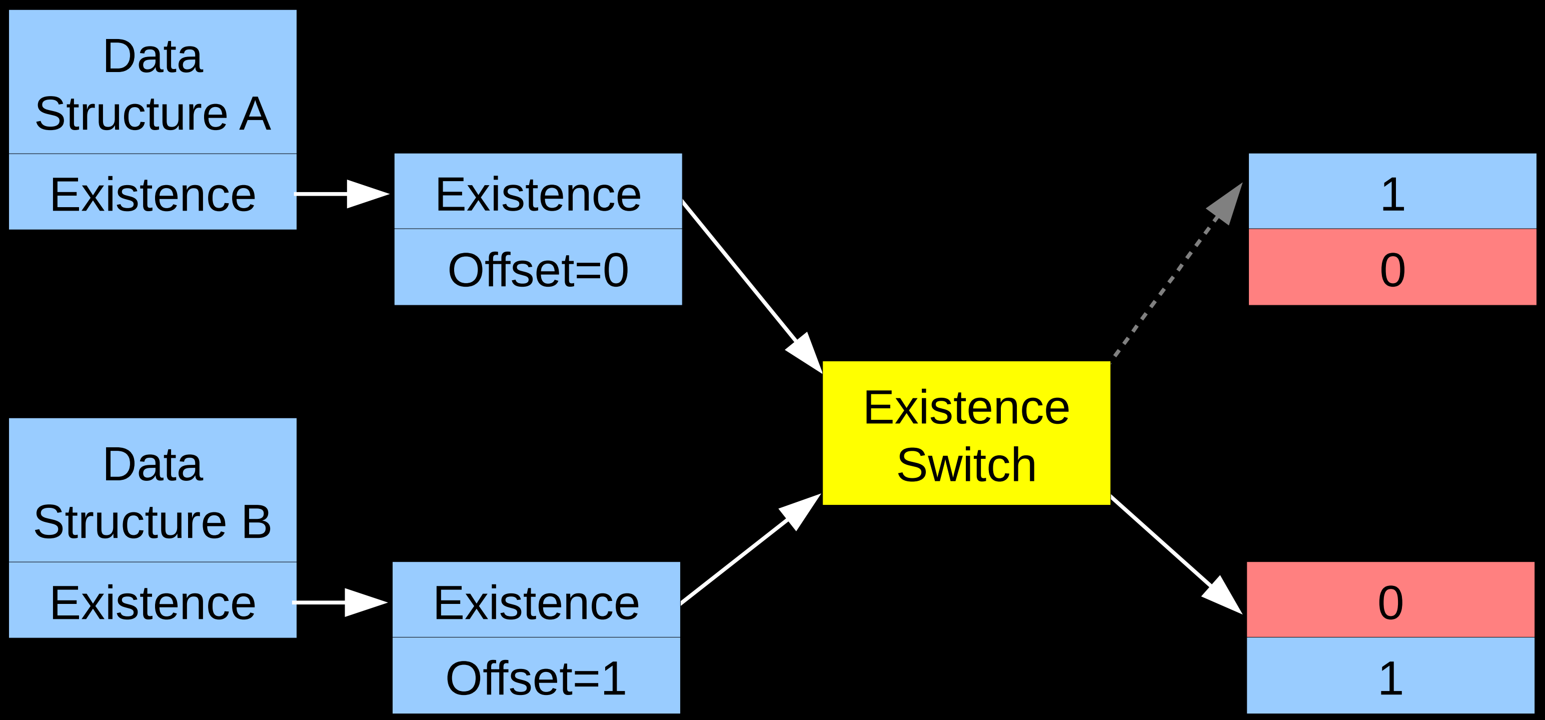
Existence Structures

- Solving yet another computer-science problem by adding an additional level of indirection...

Example Existence Structure Before Switch



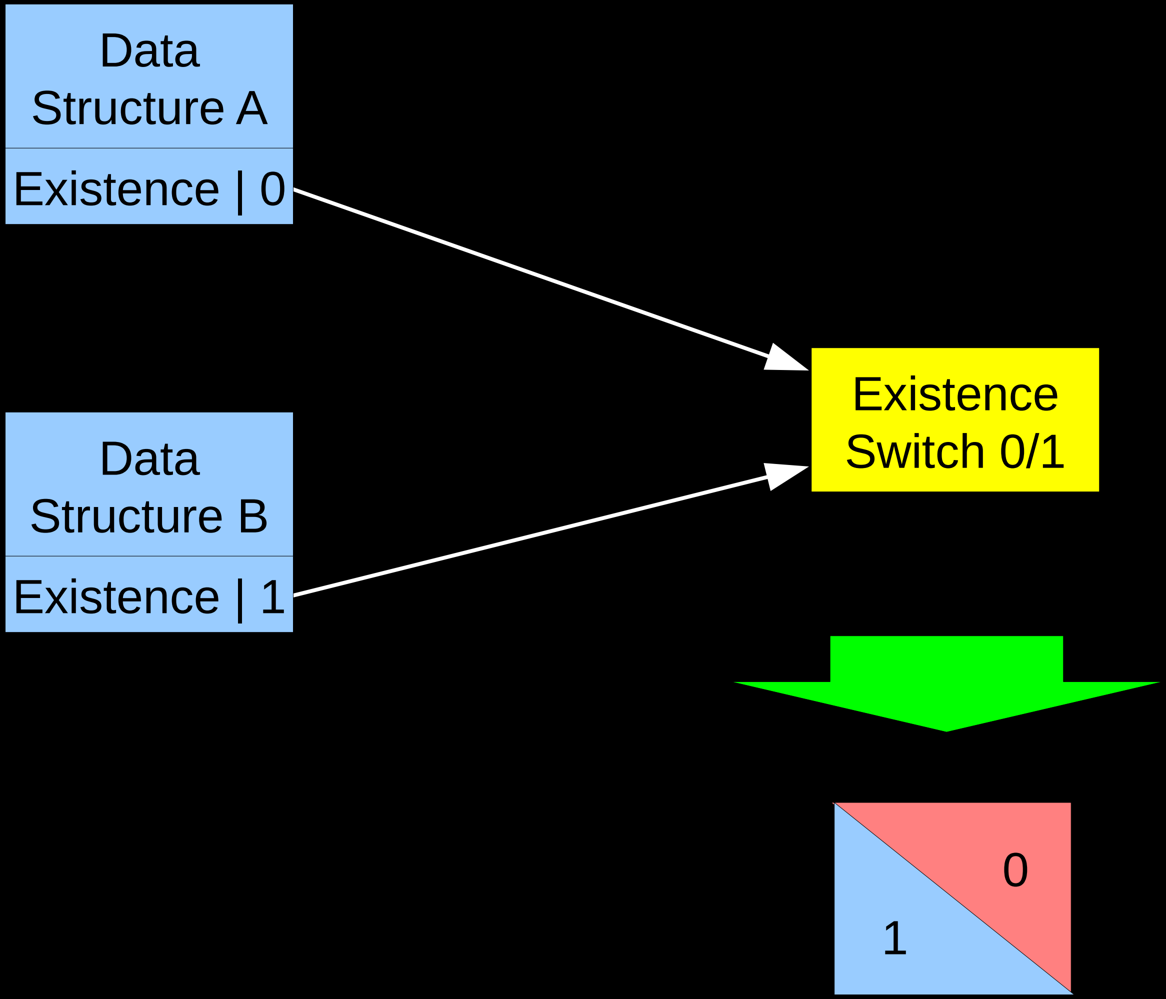
Example Existence Structure After Switch



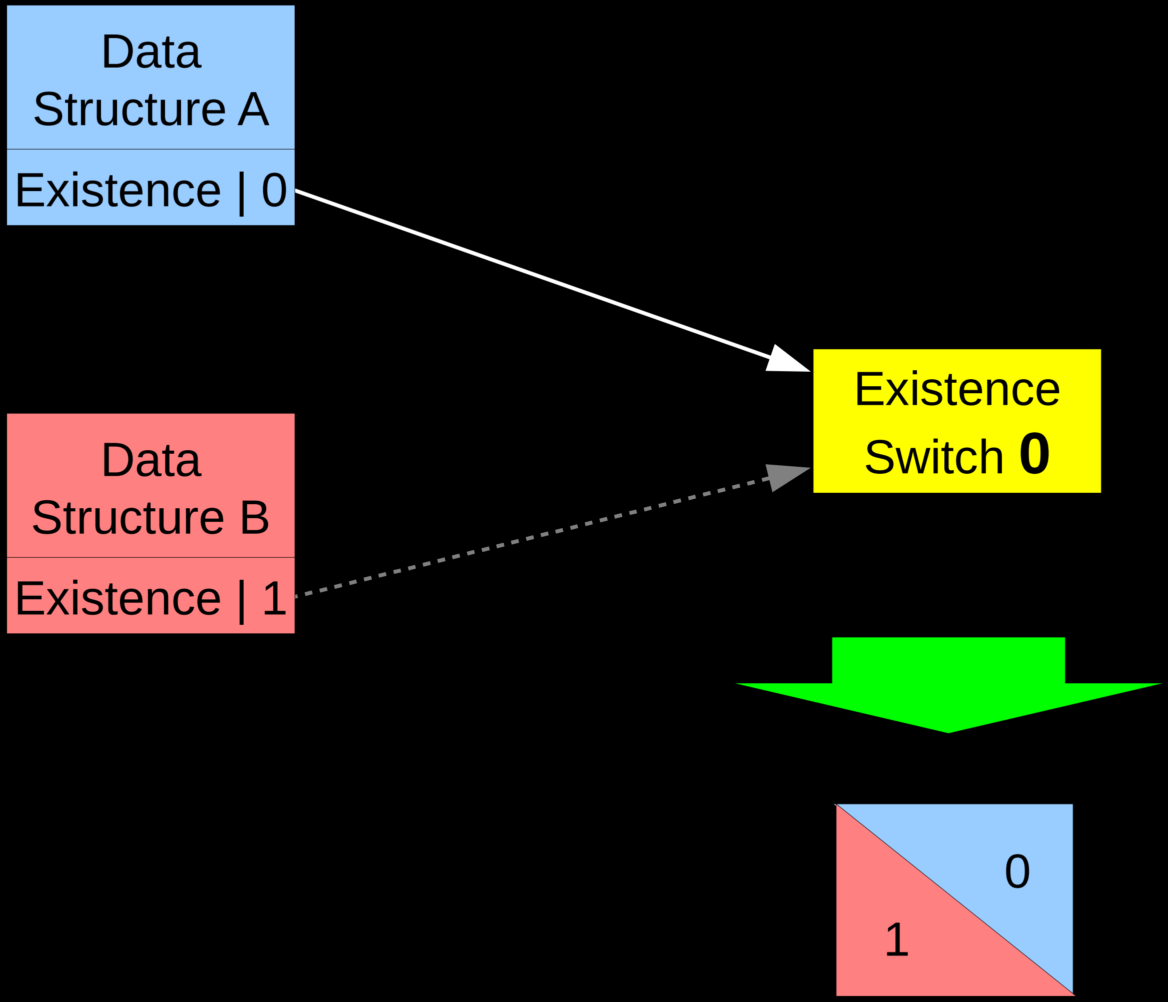
But Levels of Indirection Are Expensive!

- And I didn't just add one level of indirection, I added three!
- But most of the time, elements exist and are not being moved
- So represent this common case with a NULL pointer
  - If the existence pointer is NULL, element exists: No indirection needed
  - Backwards of the usual use of a NULL pointer, but so it goes!
- In the uncommon case, traverse existence structure as shown on the preceding slides
  - Expensive, multiple cache misses, but that is OK in the uncommon case
- There is no free lunch:
  - With this optimization, loads need smp_load_acquire() rather than READ_ONCE(), ACCESS_ONCE(), or rcu_dereference()
- Can use low-order pointer bits to remove two levels of indirection
  - Kudos to Dmitry Vyukov for this trick

Example Existence Structure: Dmitry's Approach



Example Existence Structure: Dmitry's Approach



Data Structure A

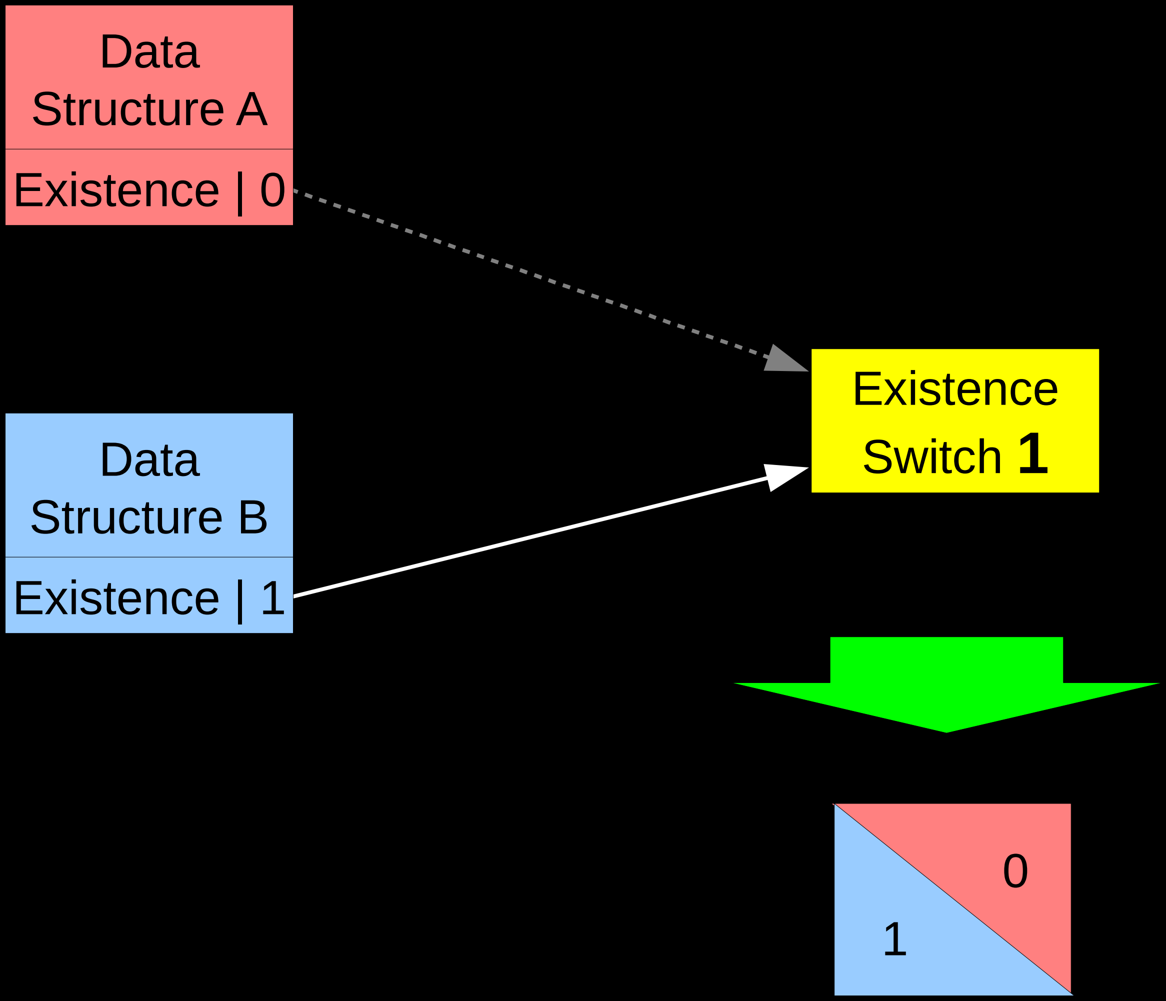
Existence | 0

Data Structure B

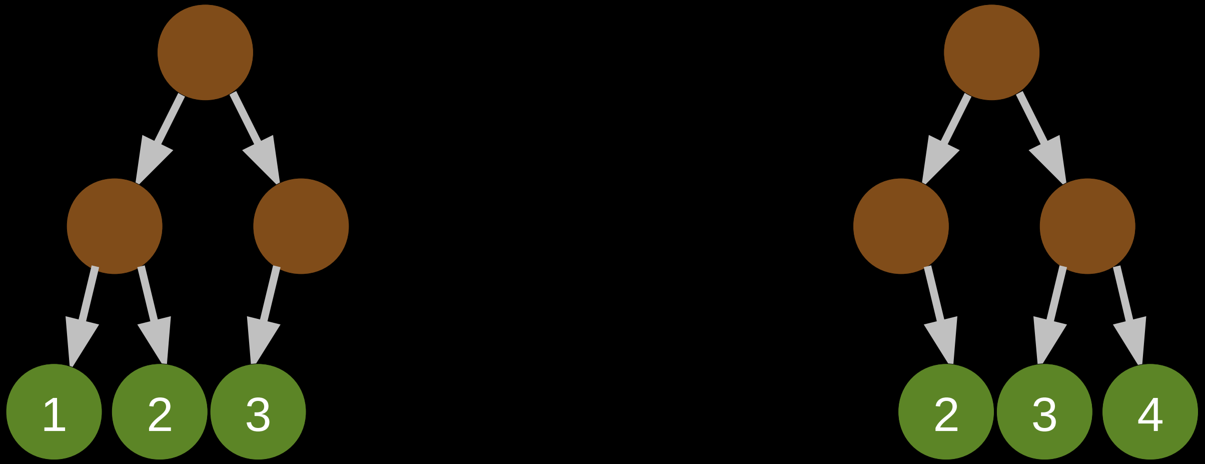
Existence | 1

Existence Switch **0**

Example Existence Structure: Dmitry's Approach

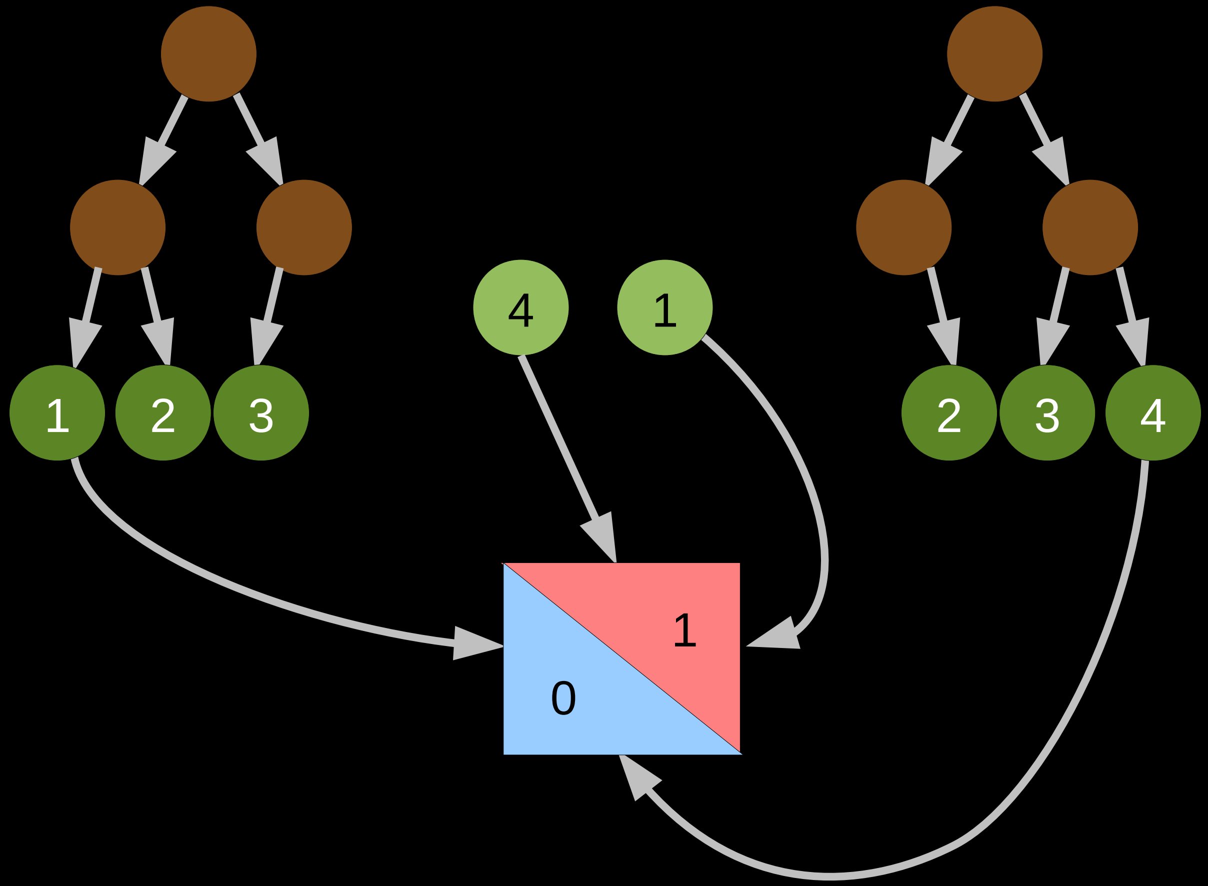


Abbreviated Existence Switch Operation (1/6)



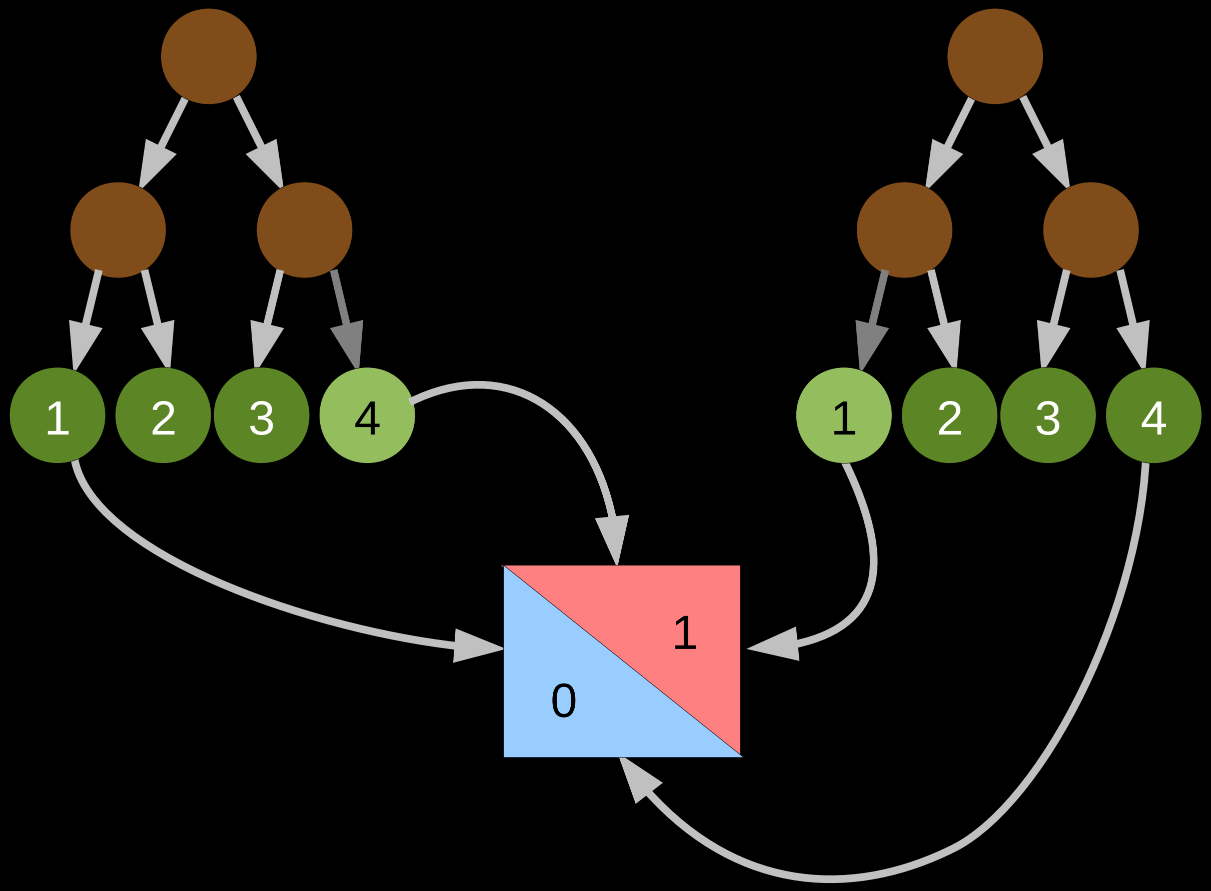
Initial state: First tree contains 1,2,3, second tree contains 2,3,4.
All existence pointers are NULL.

Abbreviated Existence Switch Operation (2/6)



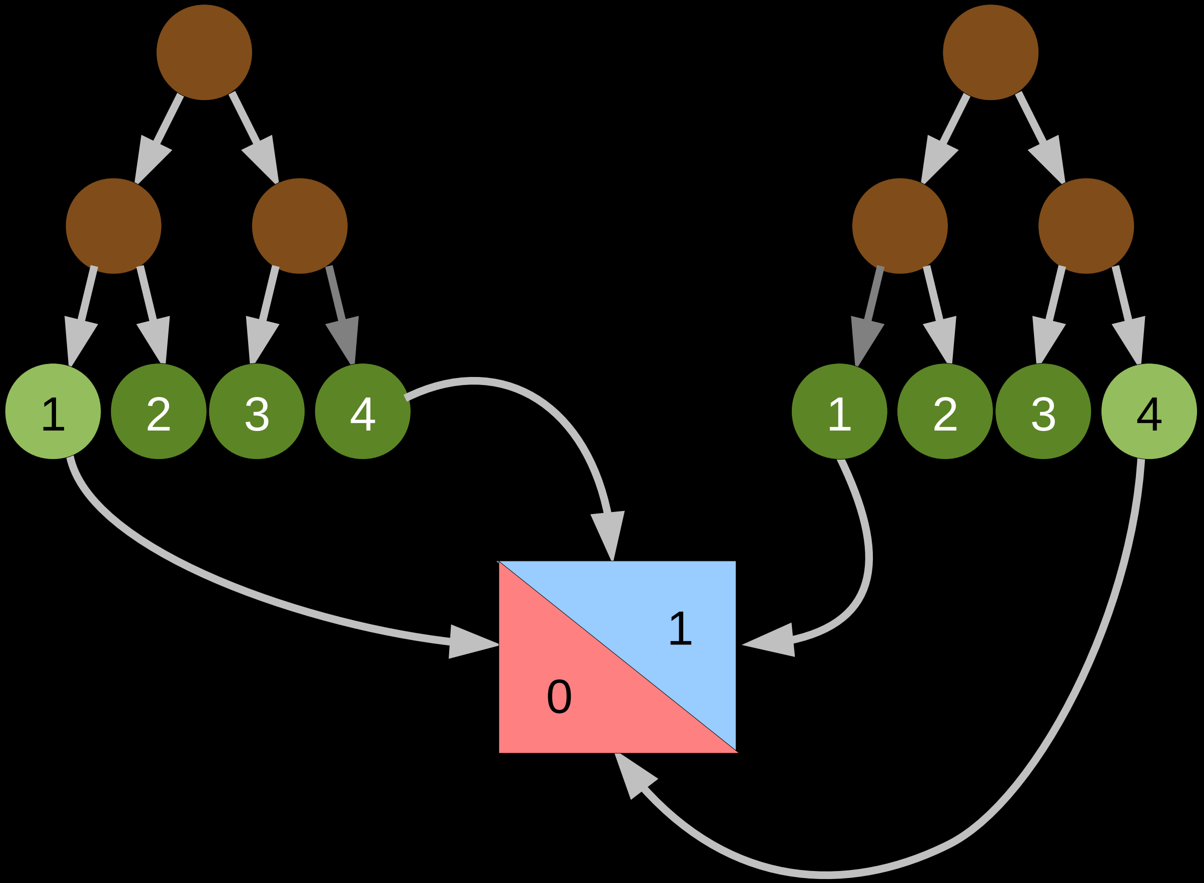
First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (3/6)



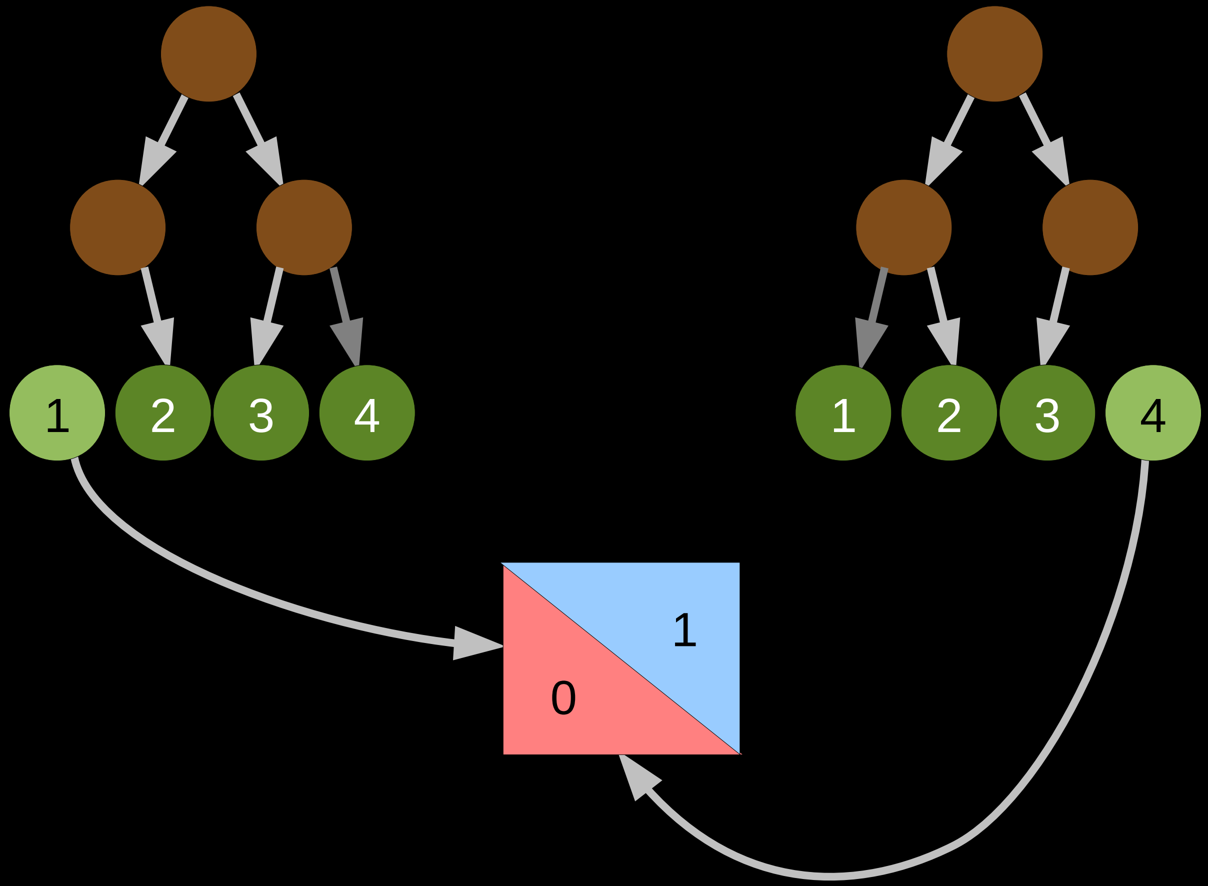
After insertion, same: First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (4/6)



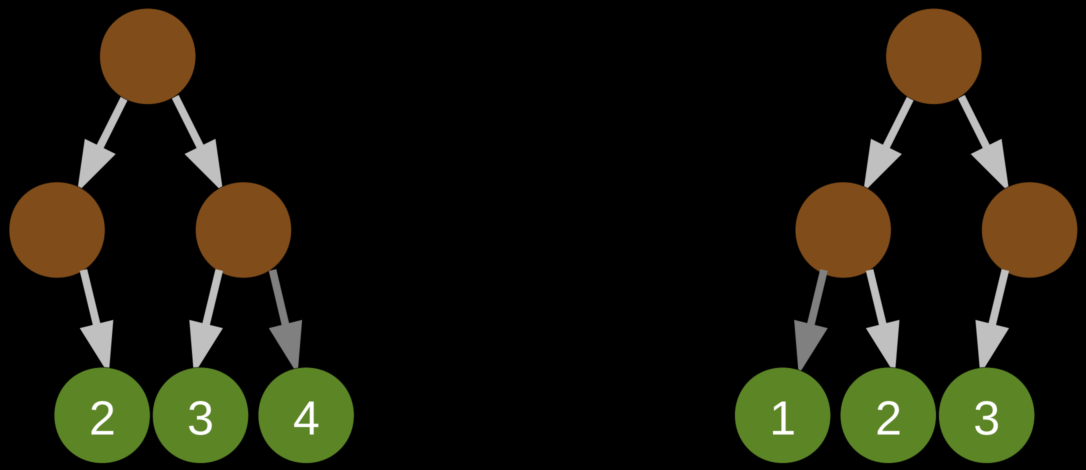
After existence switch: First tree contains 2,3,4, second tree contains 1,2,3.
Transition is single store, thus atomic! (But lookups need barriers in this case.)

Abbreviated Existence Switch Operation (5/6)



Unlink old nodes and allegiance structure

Abbreviated Existence Switch Operation (6/6)



After waiting a grace period, can free up existence structures and old nodes
And data structure preserves locality of reference!

Existence Structures

- Existence-structure reprise:
  - Each data element has an existence pointer
  - NULL pointer says “member of current structure”
  - Non-NULL pointer references an existence structure
    - Existence of multiple data elements can be switched atomically
- But this needs a good API to have a chance of getting it right!
  - Especially given that a NULL pointer means that the element exists!!!

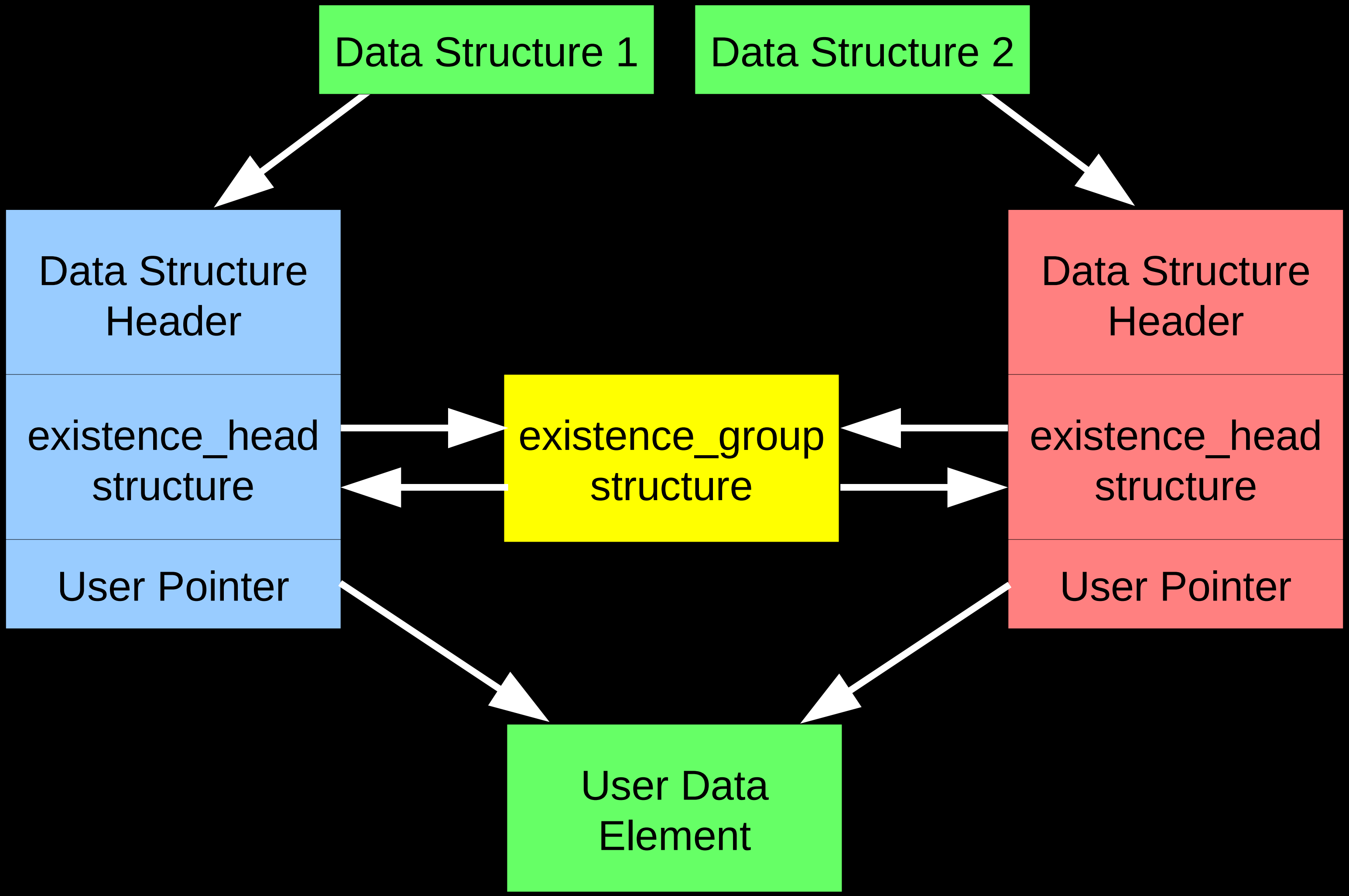
Existence Data Structures

```
struct existence_group {
        uintptr_t eg_state;
        struct cds_list_head eg_outgoing;
        struct cds_list_head eg_incoming;
        struct rcu_head eg_rh;
};

struct existence_head {
        uintptr_t eh_egi;
        struct cds_list_head eh_list;
        int (*eh_add)(struct existence_head *ehp);
        void (*eh_remove)(struct existence_head *ehp);
        void (*eh_free)(struct existence_head *ehp);
        int eh_gone;
        spinlock_t eh_lock;
        struct rcu_head eh_rh;
};
```

Existence APIs

- `void existence_init(struct existence_group *egp);`
- `uintptr_t existence_group_outgoing(struct existence_group *egp);`
- `uintptr_t existence_group_incoming(struct existence_group *egp);`
- `void existence_set(struct existence **epp, struct existence *ep);`
- `void existence_clear(struct existence **epp);`
- `int existence_exists(struct existence_head *ehp);`
- `int existence_exists_relaxed(struct existence_head *ehp);`
-
  ```
  int existence_head_init_incoming(struct existence_head *ehp,
                                   struct existence_group *egp,
                                   int (*eh_add)(struct existence_head *ehp),
                                   void (*eh_remove)(struct existence_head *ehp),
                                   void (*eh_free)(struct existence_head *ehp))
  ```
-
  ```
  int existence_head_set_outgoing(struct existence_head *ehp,
                                  struct existence_group *egp)
  ```
- `void existence_flip(struct existence_group *egp);`
- `void existence_backout(struct existence_group *egp)`

Existence Data Structures: Multiple Membership

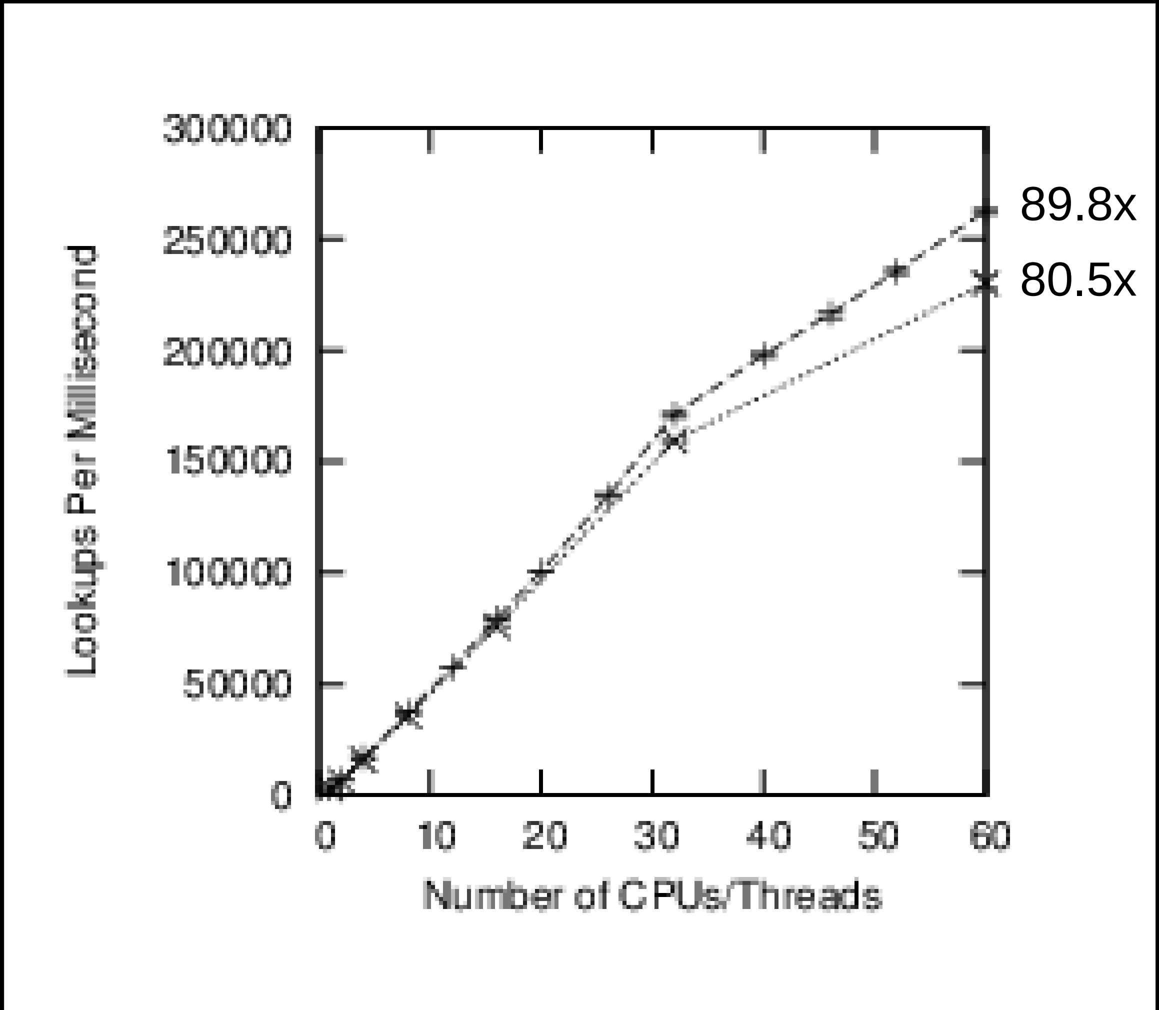


User data element atomically moving from data structure 1 to 2, which can be different types of data structures

Pseudo-Code for Atomic Move

- Allocate and initialize existence_group structure (existence_group_init())
- Add outgoing existence structure to item in source tree (existence_head_set_outgoing())
  - If operation fails, existence_backout() and report error to caller
  - Or maybe retry later
- Insert new element (with source item's data pointer) to destination tree existence_head_init_incoming())
  - If operation fails, existence_backout() and error to caller
  - Or maybe retry later
- Invoke existence_flip() to flip incoming and outgoing
  - And existence_flip() automatically cleans up after the operation
  - Just as existence_backout() does after a failed operation

Existence Structures: Performance and Scalability



89.8x LCA

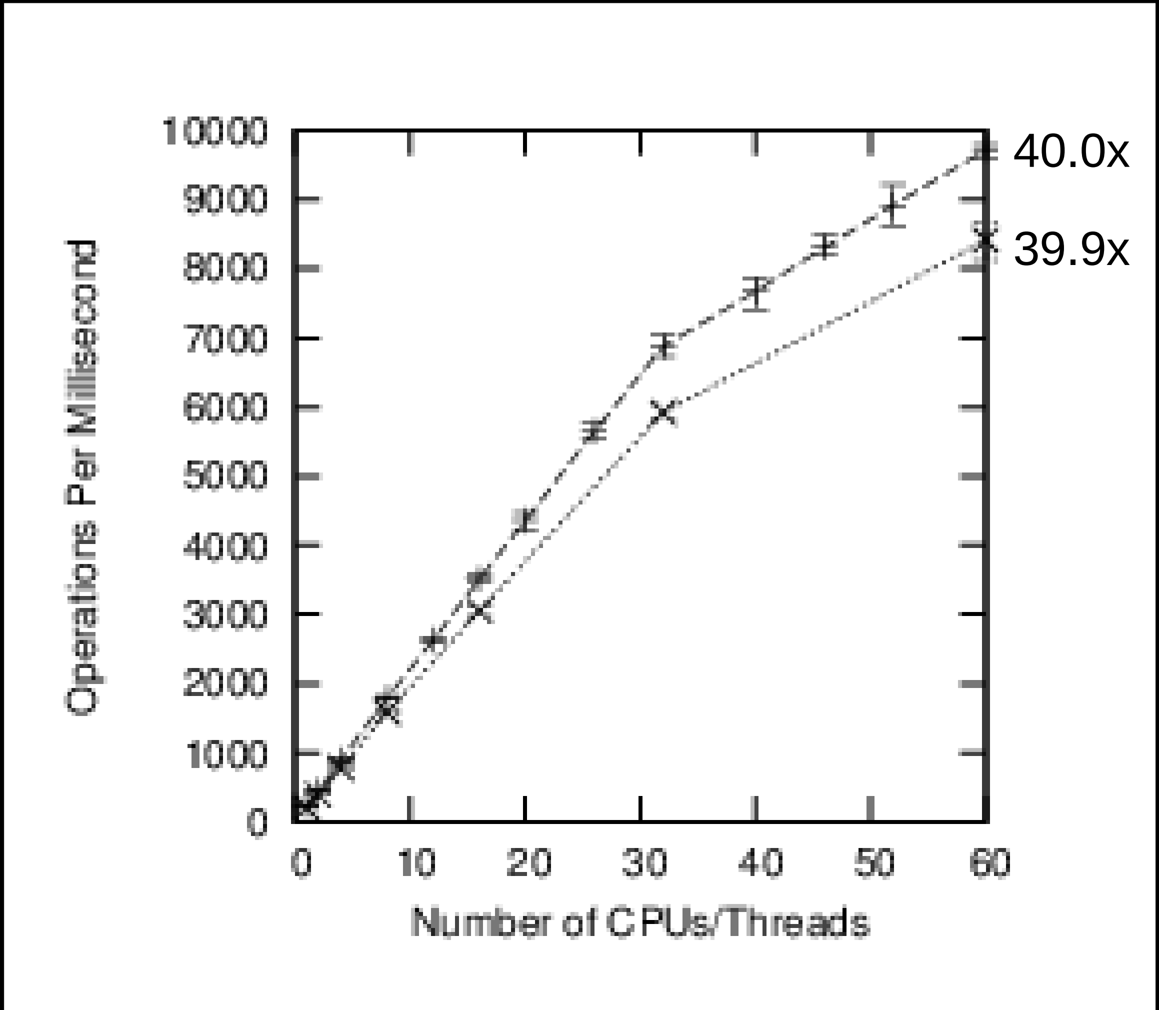
80.5x CPPCON

100% lookups

Super-linear as expected based on range partitioning

(Hash tables about 3x faster)

Existence Structures: Performance and Scalability

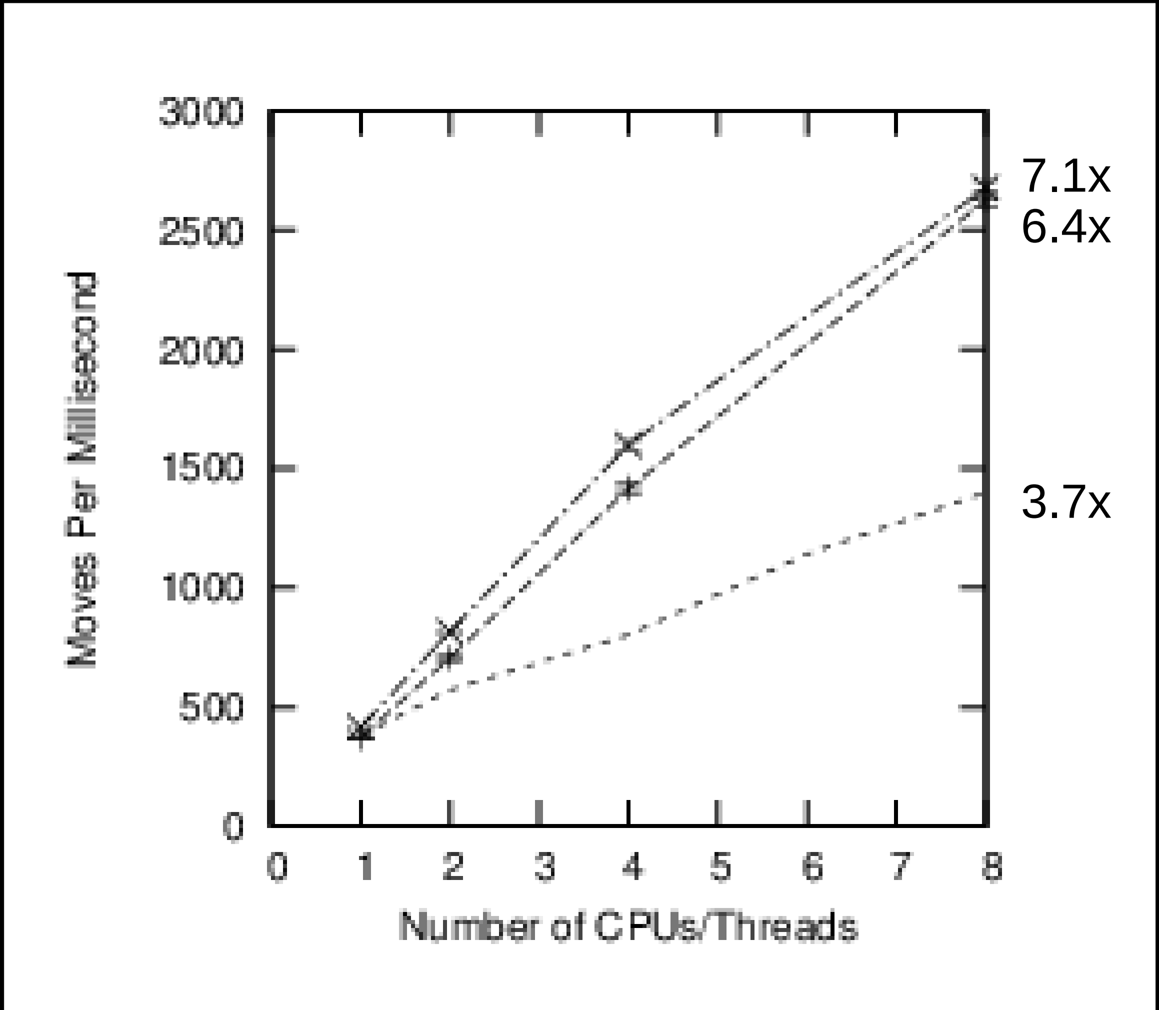


LCA

CPPCON

90% lookups, 3% insertions, 3% deletions, 3% full tree scans, 1% moves
(Workload approximates Gramoli et al. CACM Jan. 2014)

Existence Structures: Performance and Scalability



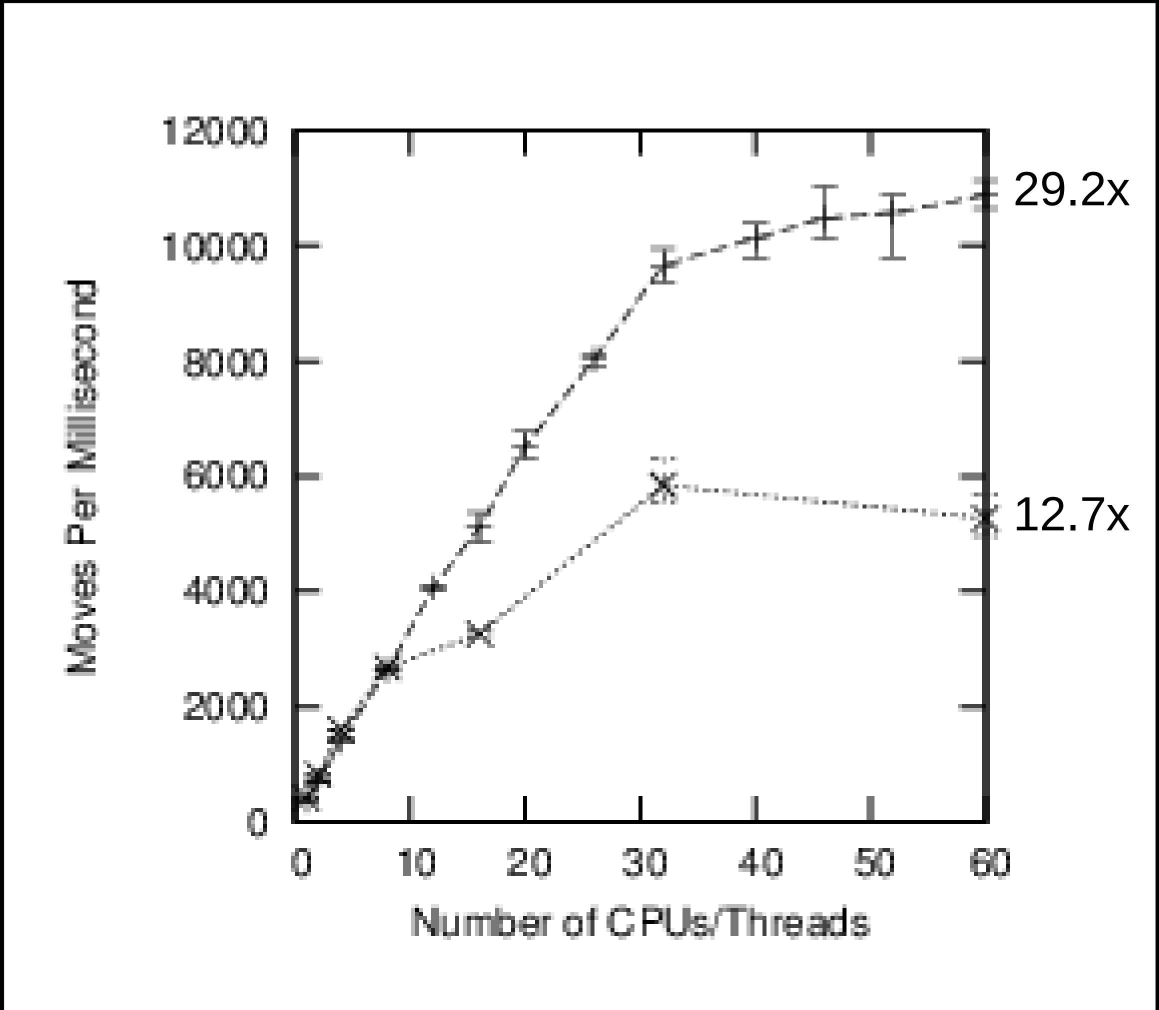
LCA

CPPCON

N4037

100% moves (worst case)

Existence Structures: Performance and Scalability



29.2x LCA

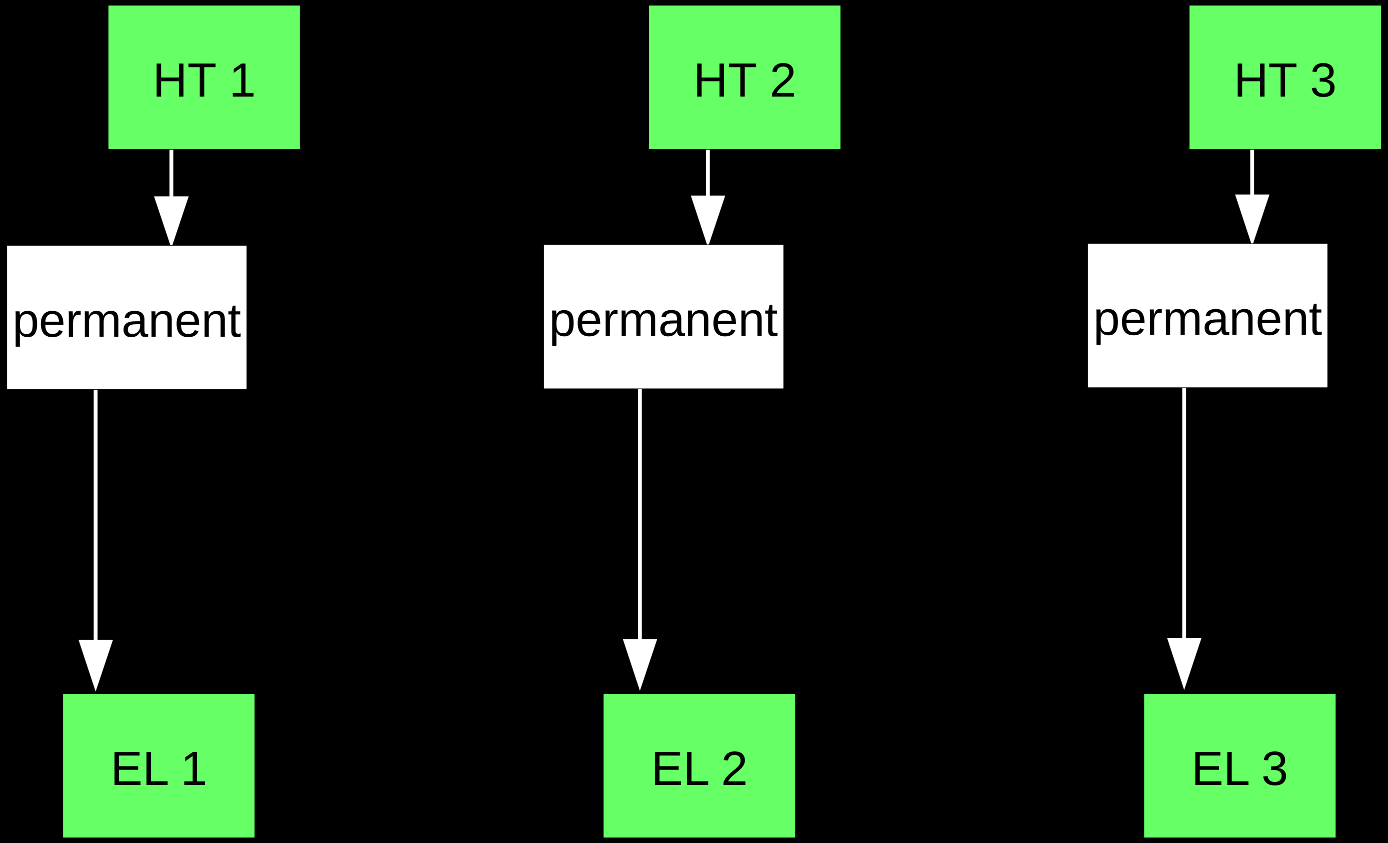
12.7x CPPCON

100% moves: Still room for improvement!

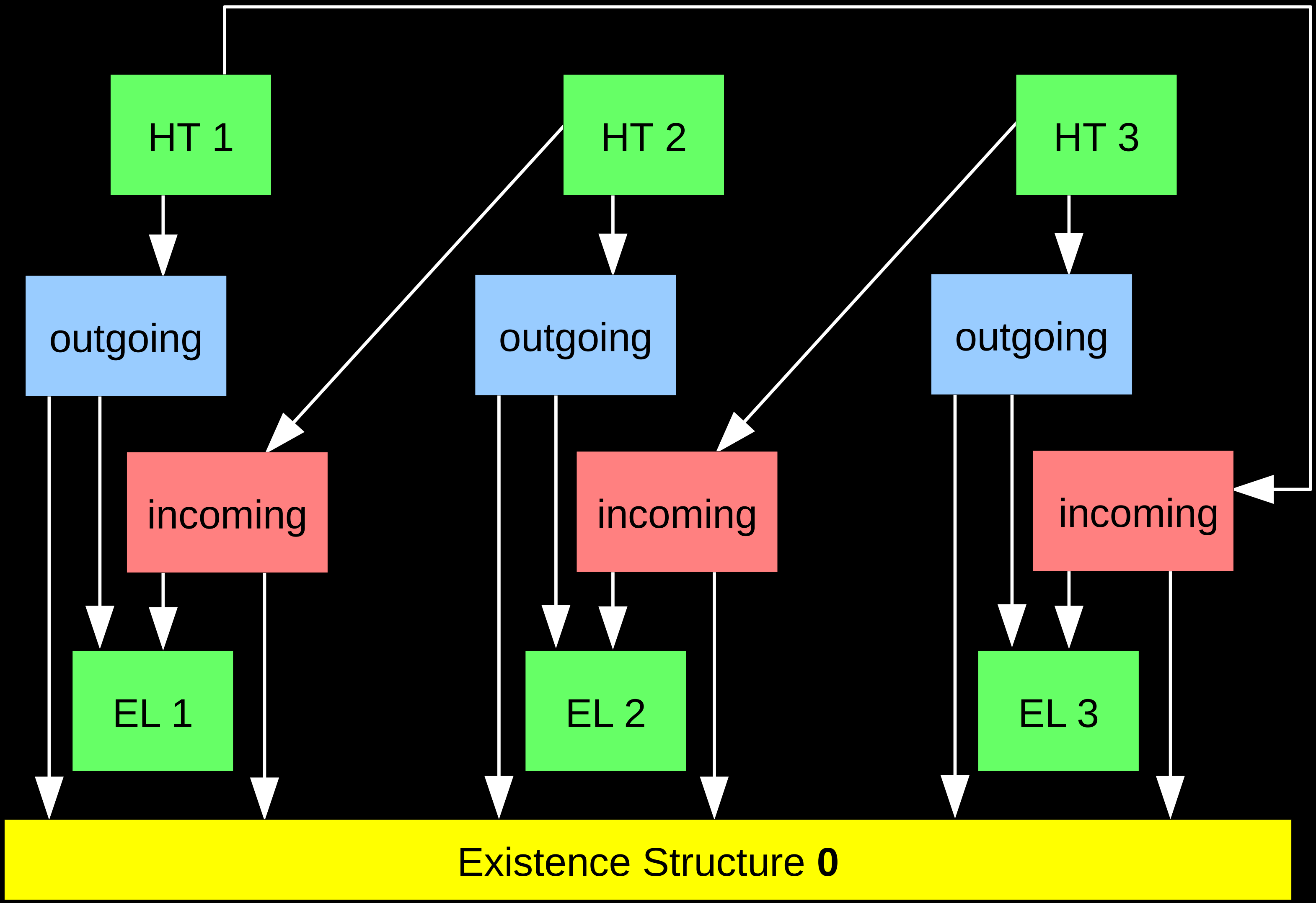
But Requires Modifications to Existing Algorithms

But Requires Modifications to Existing Algorithms New Goal: Use RCU Algorithms Unchanged!!!

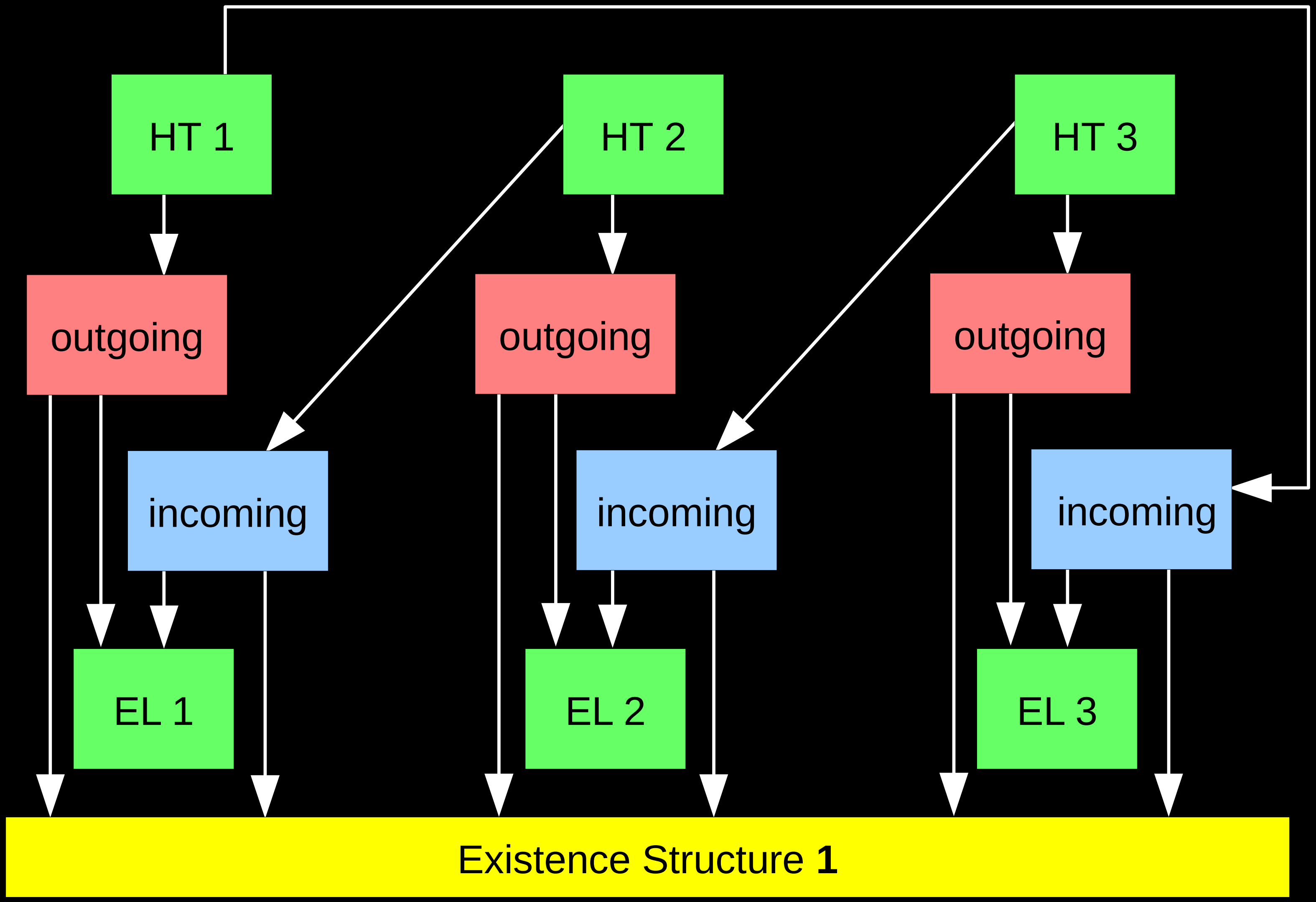
Rotate 3 Elements Through 3 Hash Tables (1/4)



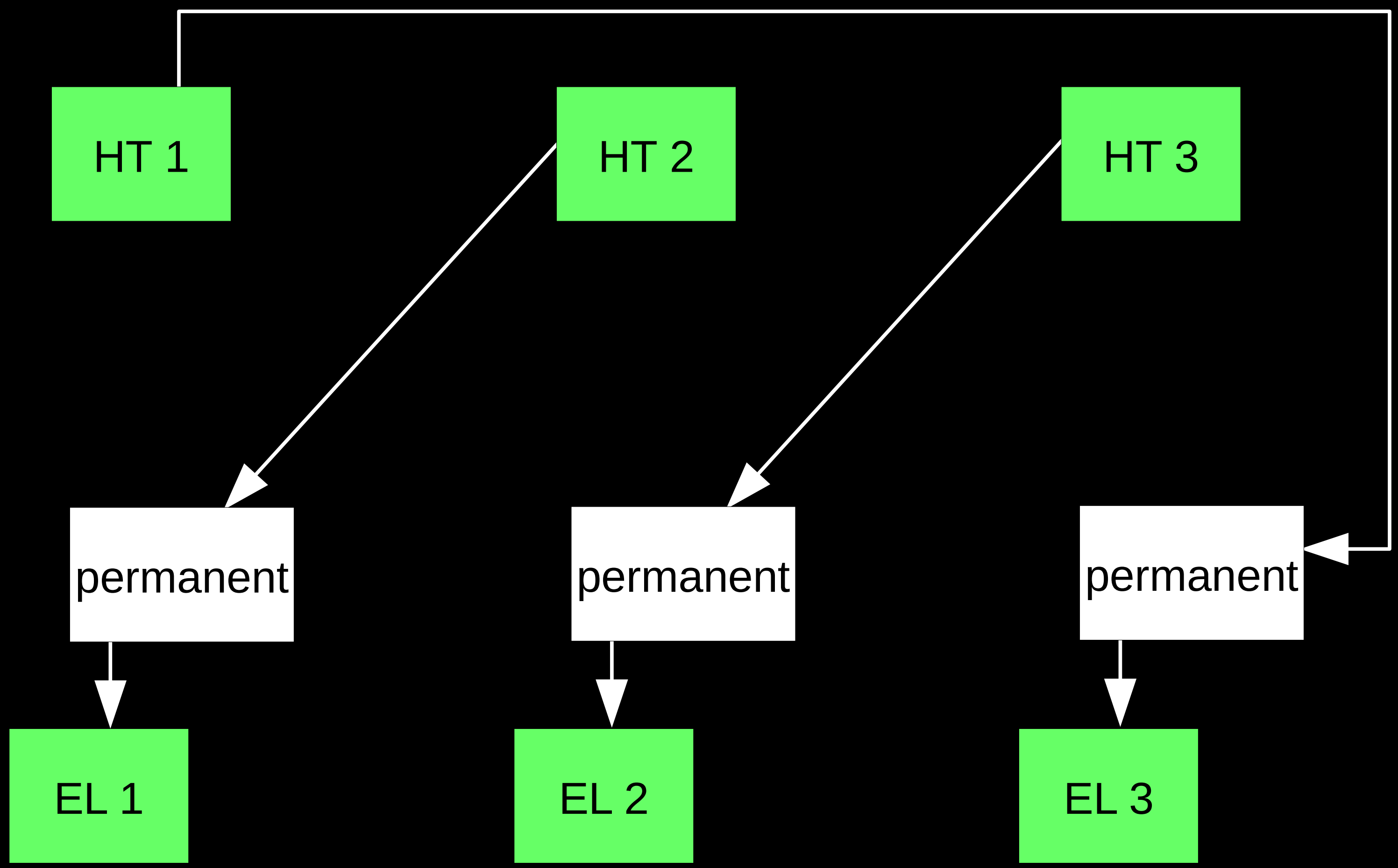
Rotate 3 Elements Through 3 Hash Tables (2/4)



Rotate 3 Elements Through 3 Hash Tables (3/4)



Rotate 3 Elements Through 3 Hash Tables (4/4)



Data to Rotate 3 Elements Through 3 Hash Tables

```
struct keyvalue {
        unsigned long key;
        unsigned long value;
        atomic_t refcnt;
};

struct hash_exists {
        struct ht_elem he_hte;
        struct hashtab *he_htp;
        struct existence_head he_eh;
        struct keyvalue *he_kv;
};
```

Code to Rotate 3 Elements Through 3 Hash Tables

```
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
heo[0] = hash_exists_alloc(egp, htp[0], hei[2]->he_kv, ~0, ~0);
heo[1] = hash_exists_alloc(egp, htp[1], hei[0]->he_kv, ~0, ~0);
heo[2] = hash_exists_alloc(egp, htp[2], hei[1]->he_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&hei[0]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[1]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[2]->he_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

BUG_ON()s become checks with calls to existence_backout() if contention possible

Code to Rotate 3 Elements Through 3 Hash Tables

```
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
heo[0] = hash_exists_alloc(egp, htp[0], hei[2]->he_kv, ~0, ~0);
heo[1] = hash_exists_alloc(egp, htp[1], hei[0]->he_kv, ~0, ~0);
heo[2] = hash_exists_alloc(egp, htp[2], hei[1]->he_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&hei[0]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[1]->he_eh, egp));
BUG_ON(existence_head_set_outgoing(&hei[2]->he_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

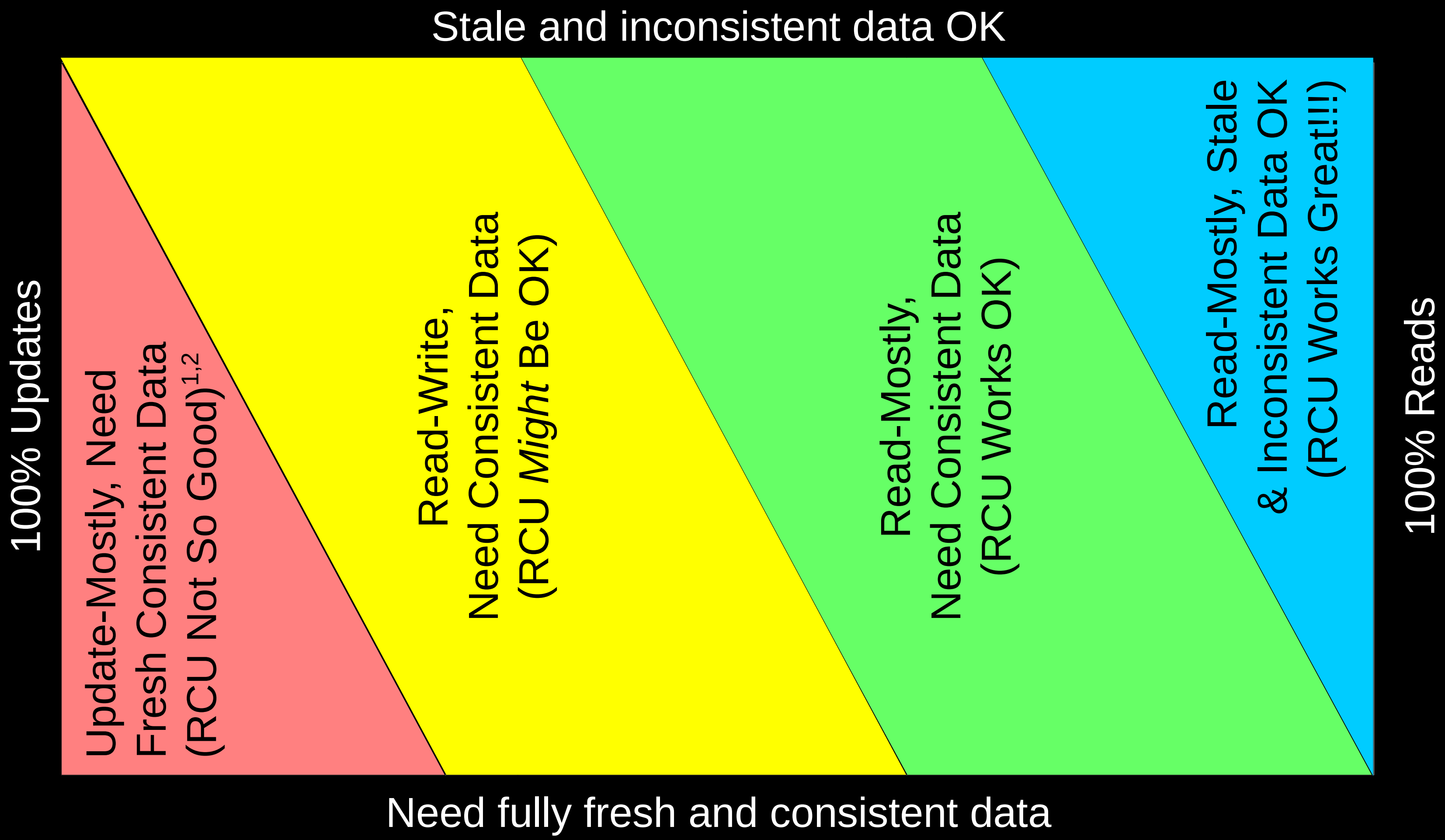
BUG_ON()s become checks with calls to existence_backout() if contention possible
Works with an RCU-protected hash table that knows nothing of atomic move!!!

Performance and Scalability of New-Age Existence Structures?

Performance and Scalability of New-Age Existence Structures?

- For readers, as good as ever
- For update-only triple-hash rotations, not so good!

Triple-Hash Rotations are Pure Updates: Red Zone!


Stale and inconsistent data OK
100% Updates
Update-Mostly, Need Fresh Consistent Data (RCU Not So Good)[1,2]
Read-Write, Need Consistent Data (RCU *Might* Be OK)
Read-Mostly, Need Consistent Data (RCU Works OK)
Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!!)
100% Reads
Need fully fresh and consistent data

Opportunity to improve the infrastructure!

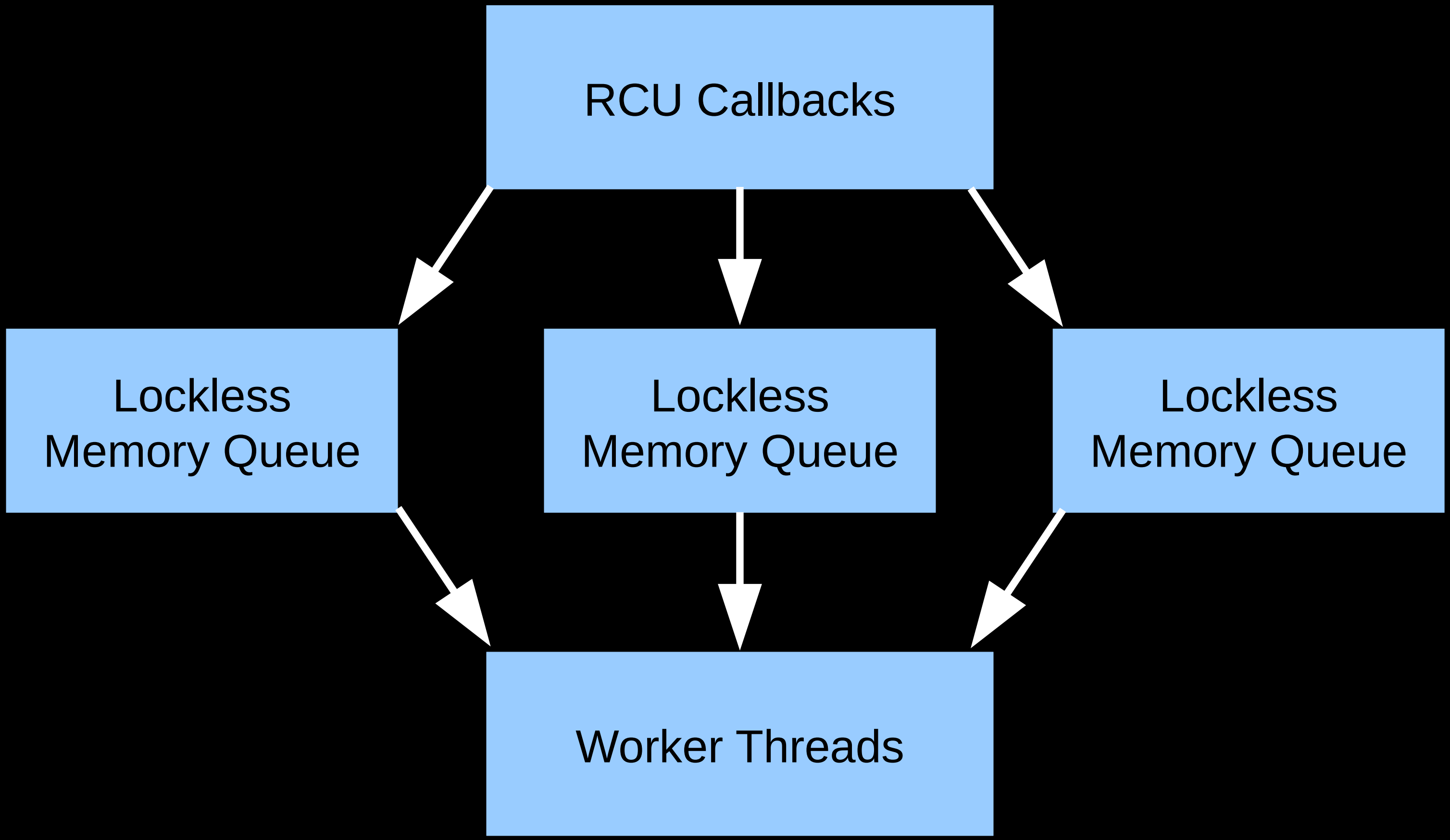
Existence Structures: Towards Update Scalability

- “Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them.” – Paraphrased from Body Heat, w/apologies to Kathleen Turner fans
- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The “perf” tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with “if (!trylock()) abort” never aborts
    - Other “perf” entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - The glibc allocator need not apply for this job
    - The jemalloc allocator bloats the per-thread lists, resulting in ever-growing RSS
    - The tcmalloc allocator suffers from lock contention moving to/from global pool
    - A tcmalloc that is better able to handle producer-consumer relations in the works, but I first heard of this a few years back and it still has not made its appearance

Existence Structures: Towards Update Scalability

- “Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them.” – Paraphrased from Body Heat, w/apologies to Kathleen Turner fans
- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The “perf” tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with “if (!trylock()) abort” never aborts
    - Other “perf” entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - The glibc allocator need not apply for this job
    - The jemalloc allocator bloats the per-thread lists, resulting in ever-growing RSS
    - The tcmalloc allocator suffers from lock contention moving to/from global pool
    - A tcmalloc that is better able to handle producer-consumer relations in the works, but I first heard of this a few years back and it still has not made its appearance
- Fortunately, I have long experience with memory allocators
  - McKenney & Slingwine, “Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors”, 1993 USENIX
  - But needed to complete implementation in one day, so chose quick hack

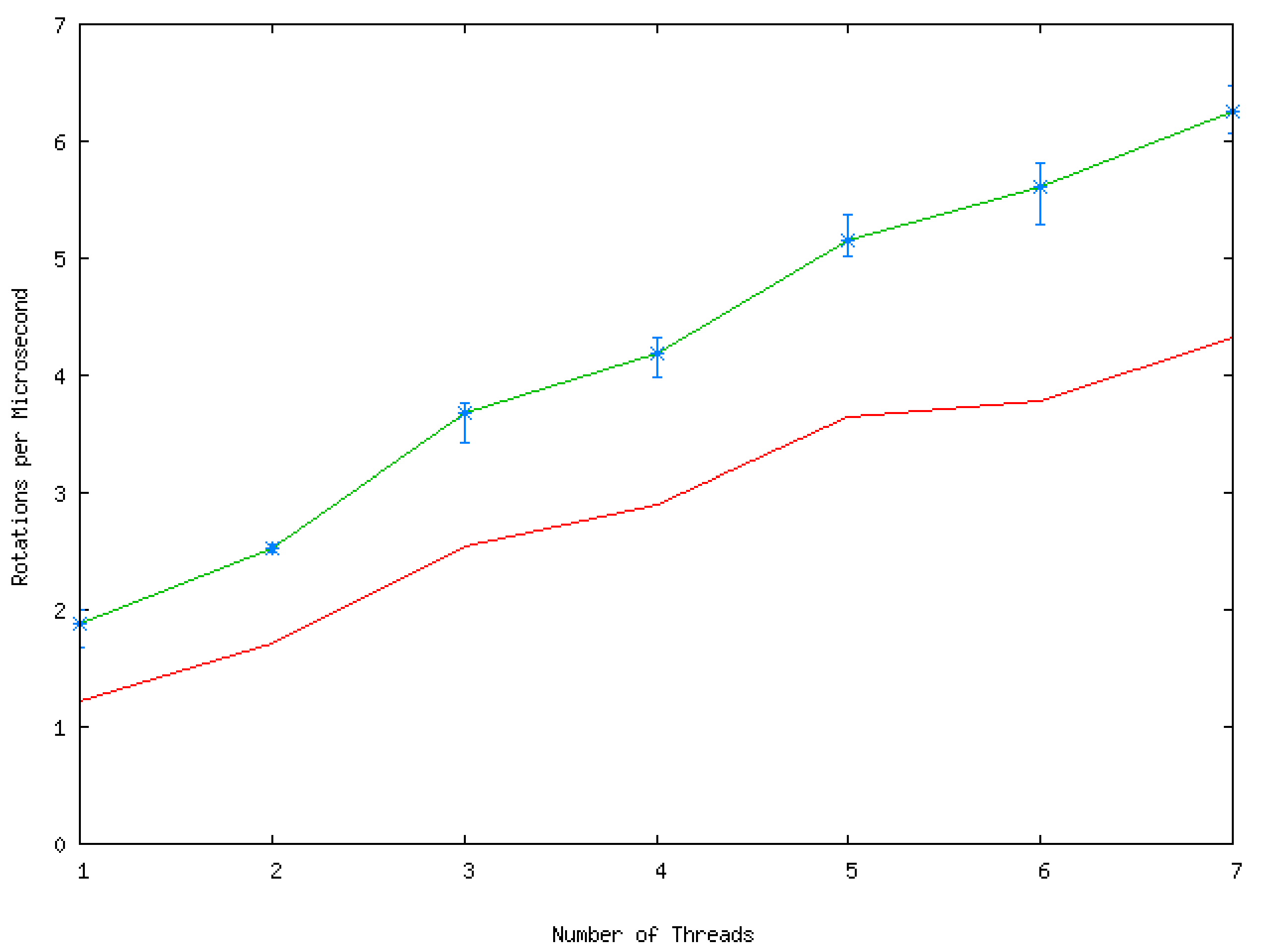
Specialized Producer/Consumer Allocator



New Age Existence Structures: Towards Scalability

- “Providing perfect performance and scalability is like committing the perfect crime. There are 50 things that might go wrong, and if you are a genius, you might be able to foresee and forestall 25 of them.” – Paraphrased from Body Heat, with apologies to Kathleen Turner fans
- Issues thus far:
  - Data structure alignment (false sharing) – easy fix
  - User-space RCU configuration (need per-thread call_rcu() handling, also easy fix)
  - The “perf” tool shows massive futex contention, checking locking design finds nothing
    - And replacing all lock acquisitions with “if (!trylock()) abort” never aborts
    - Other “perf” entries shift suspicion to memory allocators
  - Non-scalable memory allocators: More complex operations means more allocations!!!
    - Lockless memory queue greatly reduces memory-allocator lock contention
  - Profiling shows increased memory footprint is an issue: caches and TLBs!
  - Userspace RCU callback handling appears to be the next bottleneck
    - Perhaps some of techniques from the Linux kernel are needed in userspace

Performance and Scalability of New-Age Existence Structures for Triple Hash Rotation?



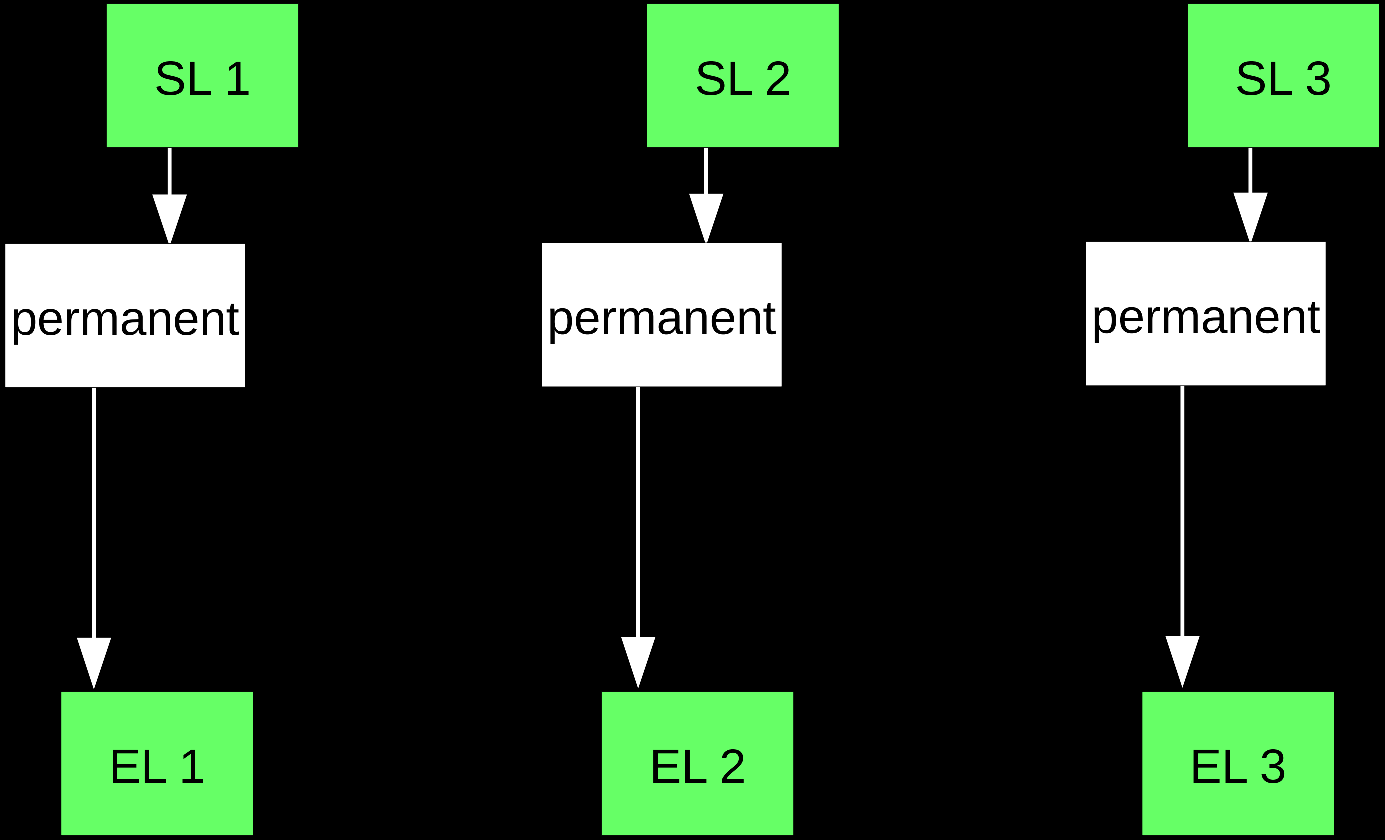
CPPCON

ACM App.

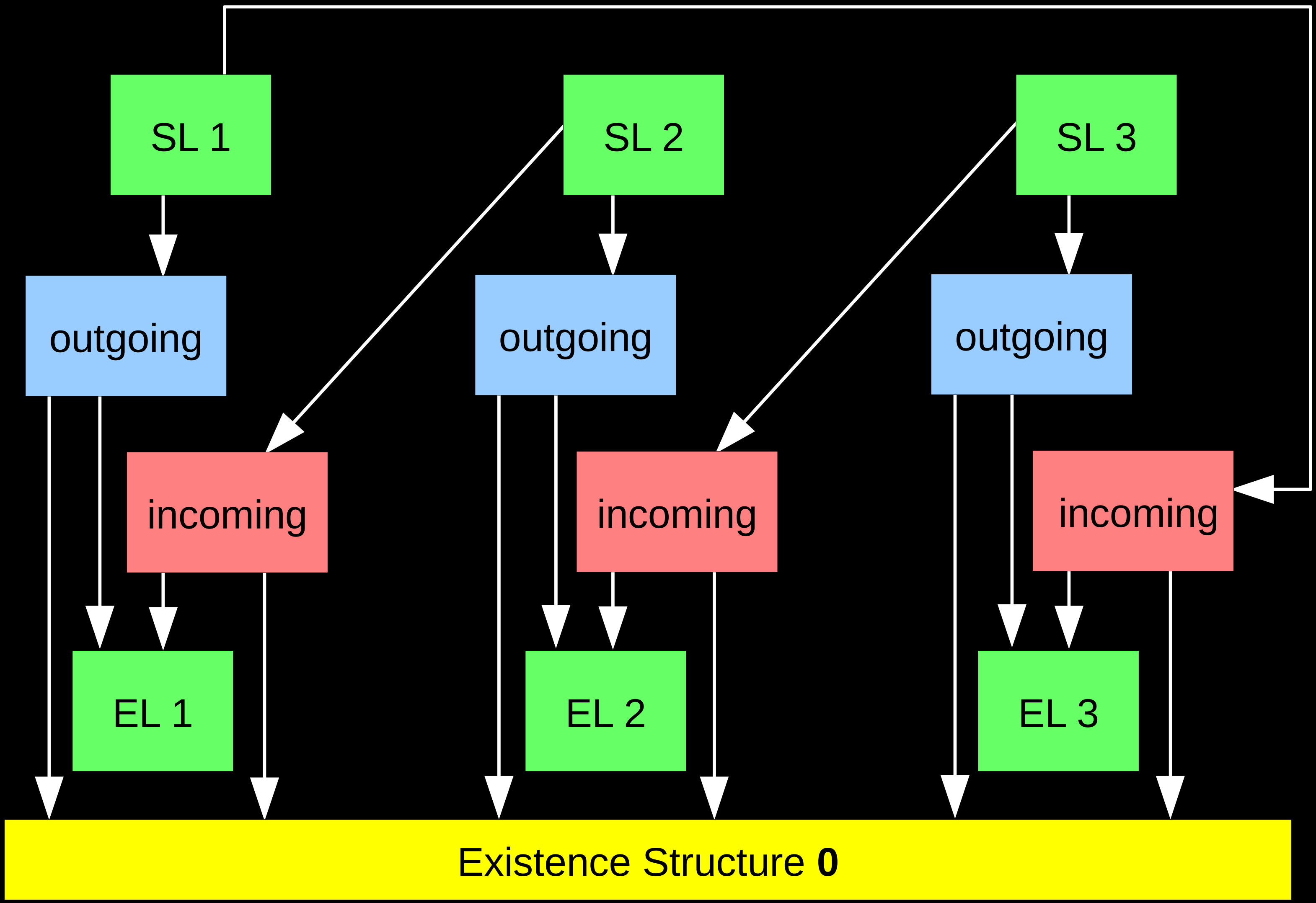
Some improvement, but still not spectacular
But note that each thread is rotating concurrently

But What About Skiplists?

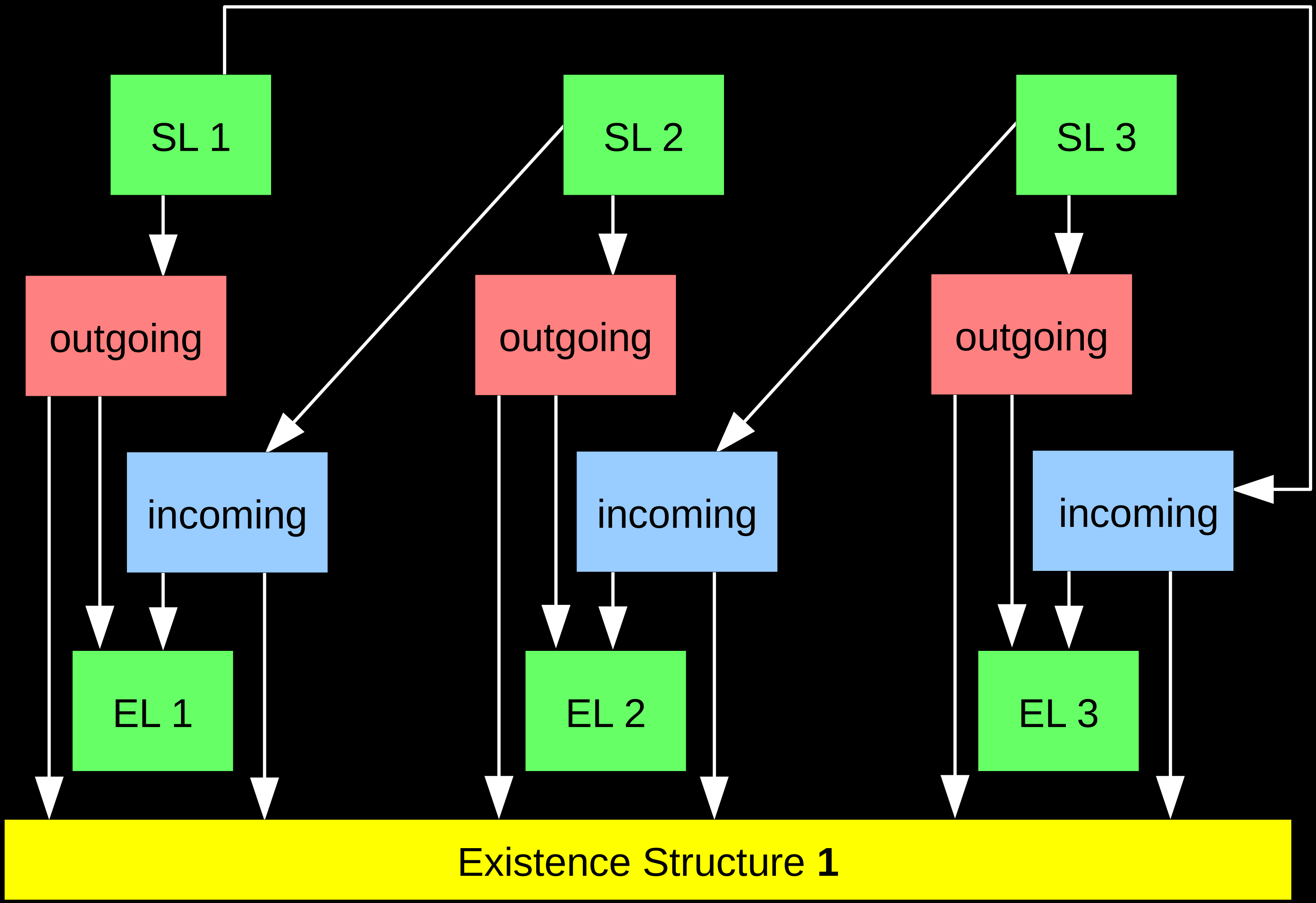
Rotate 3 Elements Through 3 Skiplists (1/4)



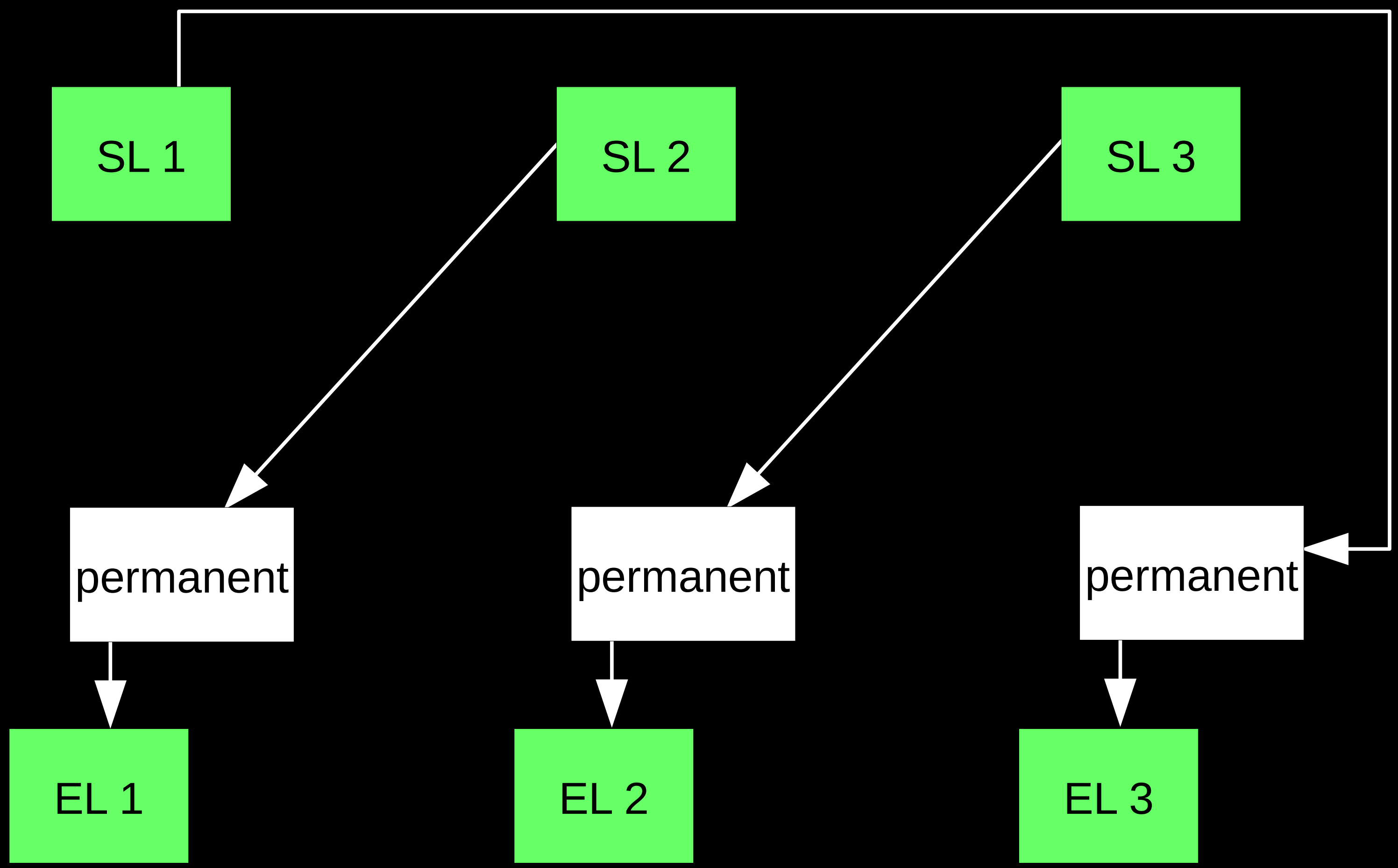
Rotate 3 Elements Through 3 Skiplists (2/4)



Rotate 3 Elements Through 3 Skiplists (3/4)



Rotate 3 Elements Through 3 Skiplists (4/4)



Data to Rotate 3 Elements Through 3 Skiplists

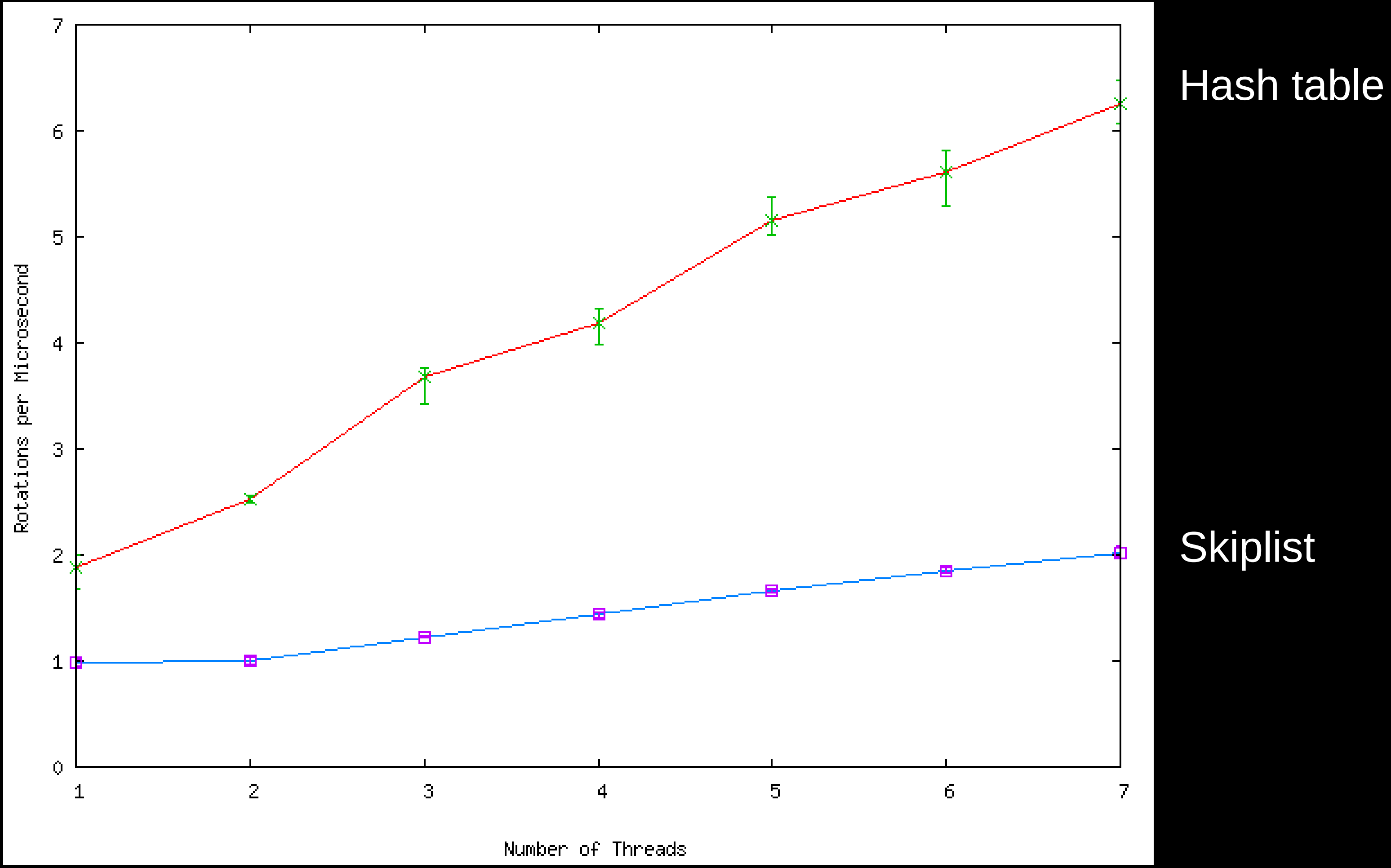
```
struct keyvalue {
        unsigned long key;
        unsigned long value;
        atomic_t refcnt;
};

struct hash_exists {
        struct skiplist se_sle;
        struct skiplist *se_slh;
        struct existence_head se_eh;
        struct keyvalue *se_kv;
};
```

Code to Rotate 3 Elements Through 3 Skiplists

```
egp = malloc(sizeof(*egp));
BUG_ON(!egp);
existence_group_init(egp);
rcu_read_lock();
seo[0] = skiplist_exists_alloc(egp, &slp[0], sei[2]->se_kv, ~0, ~0);
seo[1] = skiplist_exists_alloc(egp, &slp[1], sei[0]->se_kv, ~0, ~0);
seo[2] = skiplist_exists_alloc(egp, &slp[2], sei[1]->se_kv, ~0, ~0);
BUG_ON(existence_head_set_outgoing(&sei[0]->se_eh, egp));
BUG_ON(existence_head_set_outgoing(&sei[1]->se_eh, egp));
BUG_ON(existence_head_set_outgoing(&sei[2]->se_eh, egp));
rcu_read_unlock();
existence_flip(egp);
call_rcu(&egp->eg_rh, existence_group_rcu_cb);
```

As with hash table:RCU-protected skiplist that knows nothing of atomic move

Performance and Scalability of New-Age Existence Structures for Triple Skiplist Rotation?



This skiplist is a random tree, so we have lock contention

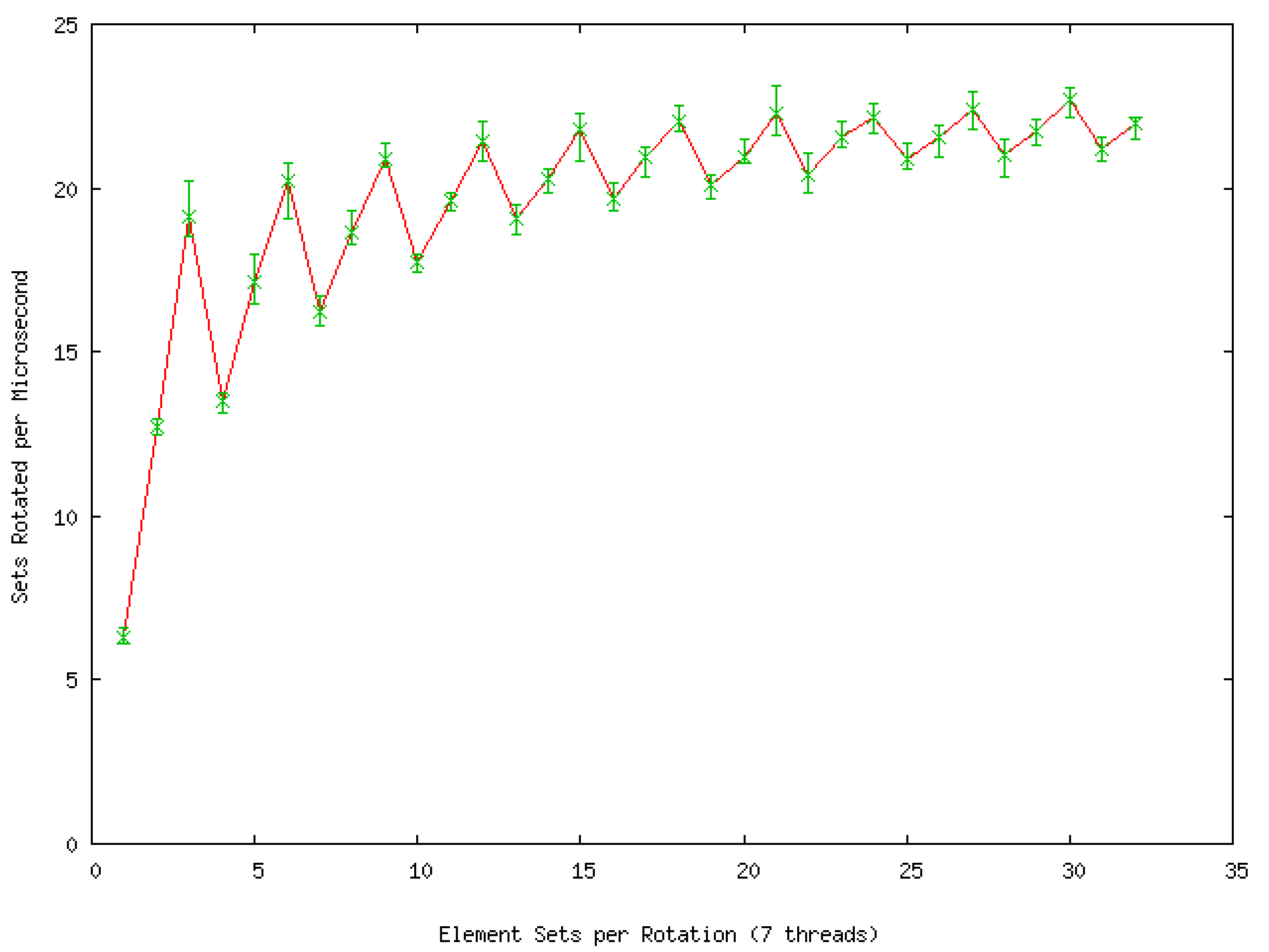
But Can We Atomically Rotate More Elements?

- Apply batching optimization!
- Instead of rotating three elements through three hash tables, rotate three pairs of elements
- Then three triplets of elements
- And so on, rotating ever larger sets through the three tables

But Can We Atomically Rotate More Elements?

- Apply batching optimization!
- Instead of rotating three elements through three hash tables, rotate three pairs of elements
- Then three triplets of elements
- And so on, rotating ever larger sets through the three tables
- It can be done, but there is a performance mystery

Large-Hash-Rotation Performance Mystery



Many additional optimizations are possible, but...

Even Bigger Mystery: Why Rotate This Way???

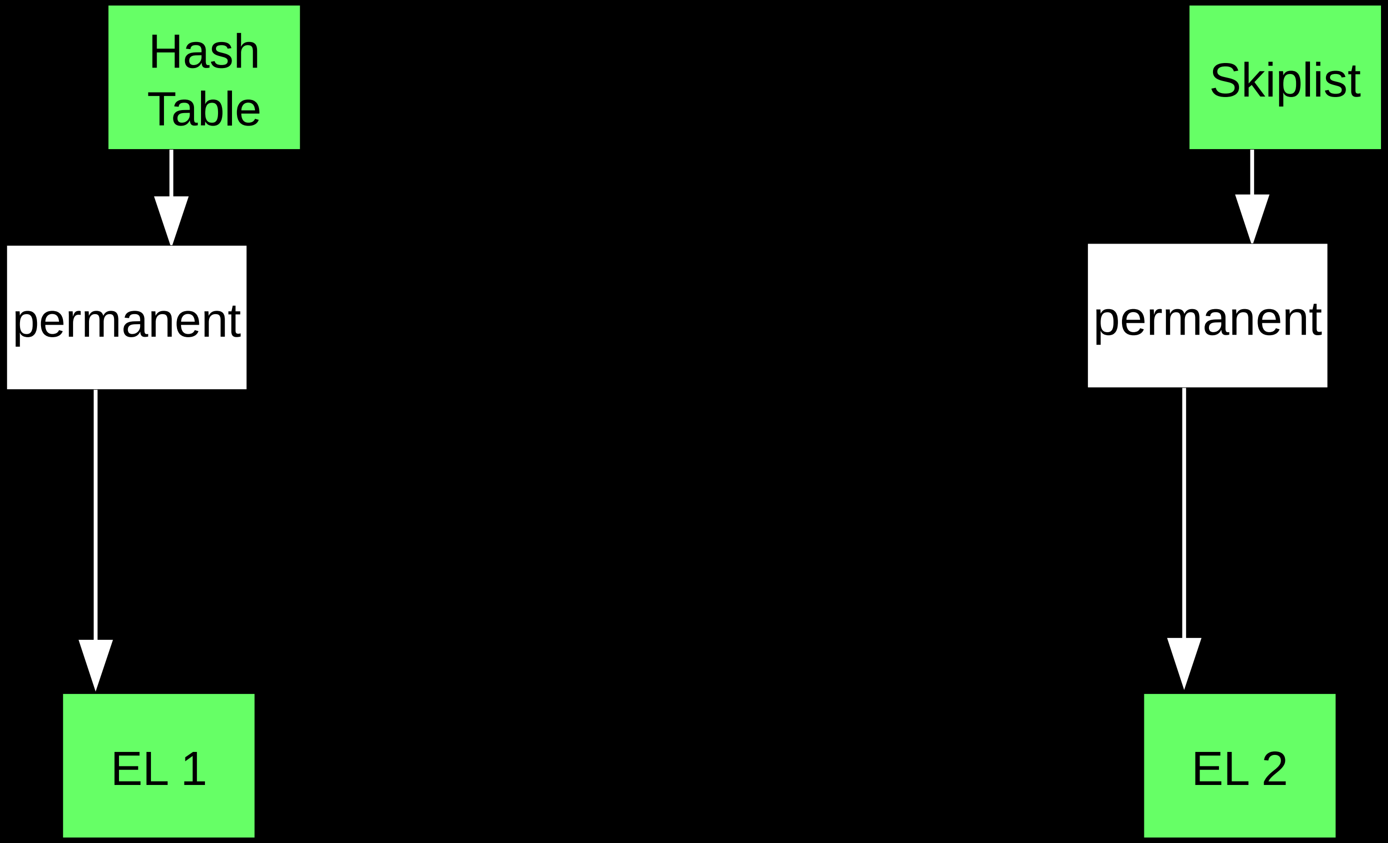
Even Bigger Mystery: Why Rotate This Way???

- Every third rotation brings us back to the original state
- So why bother with allocation, freeing, and grace periods?

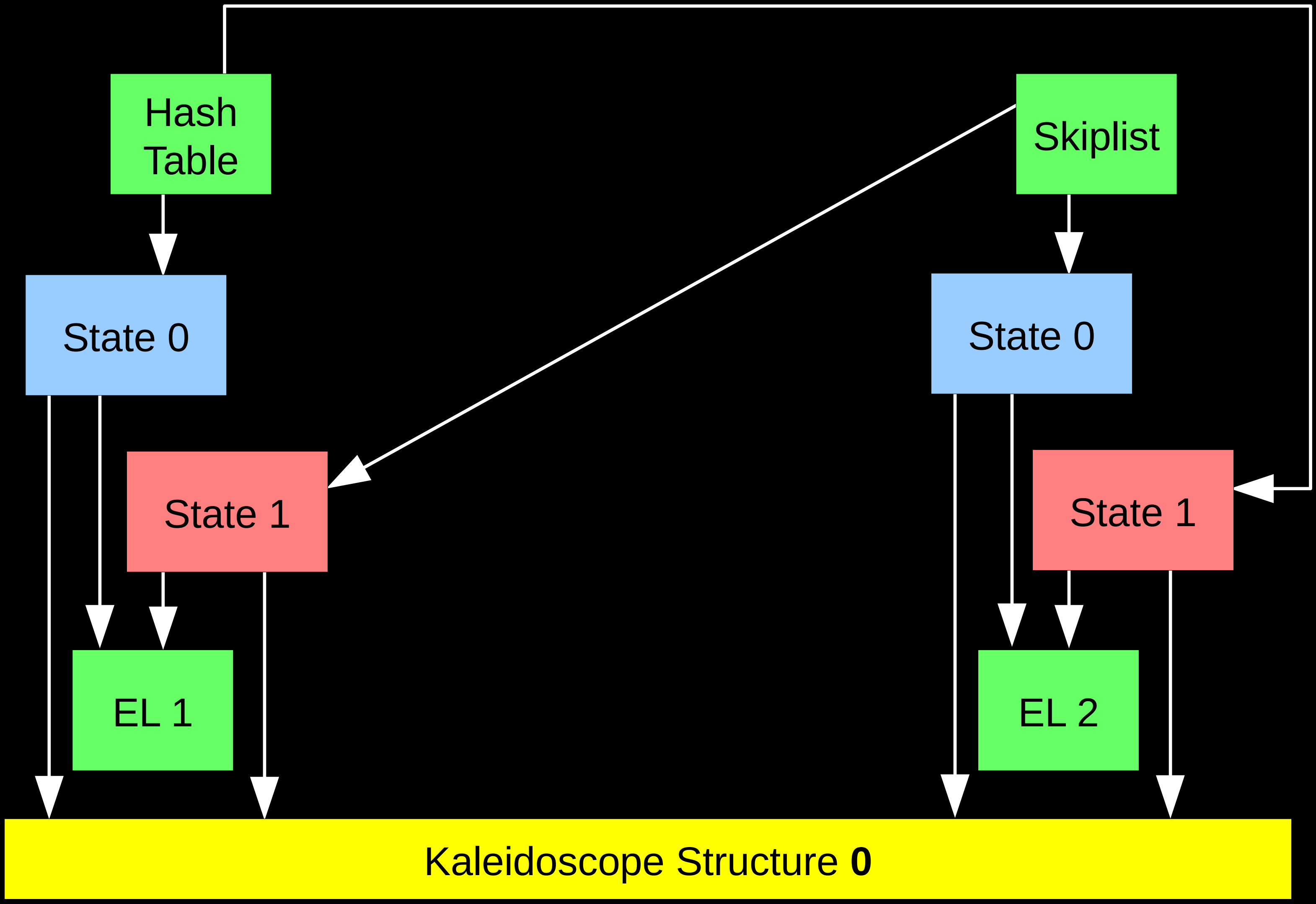
Even Bigger Mystery: Why Rotate This Way???

- Every third rotation brings us back to the original state
- So why bother with allocation, freeing, and grace periods?
- Just change the existence state variable!!!
  - But we need not be limited to two states
  - Define *kaleidoscopic data structure* as one updated by state change
  - Data structures and algorithms are very similar to those for existence

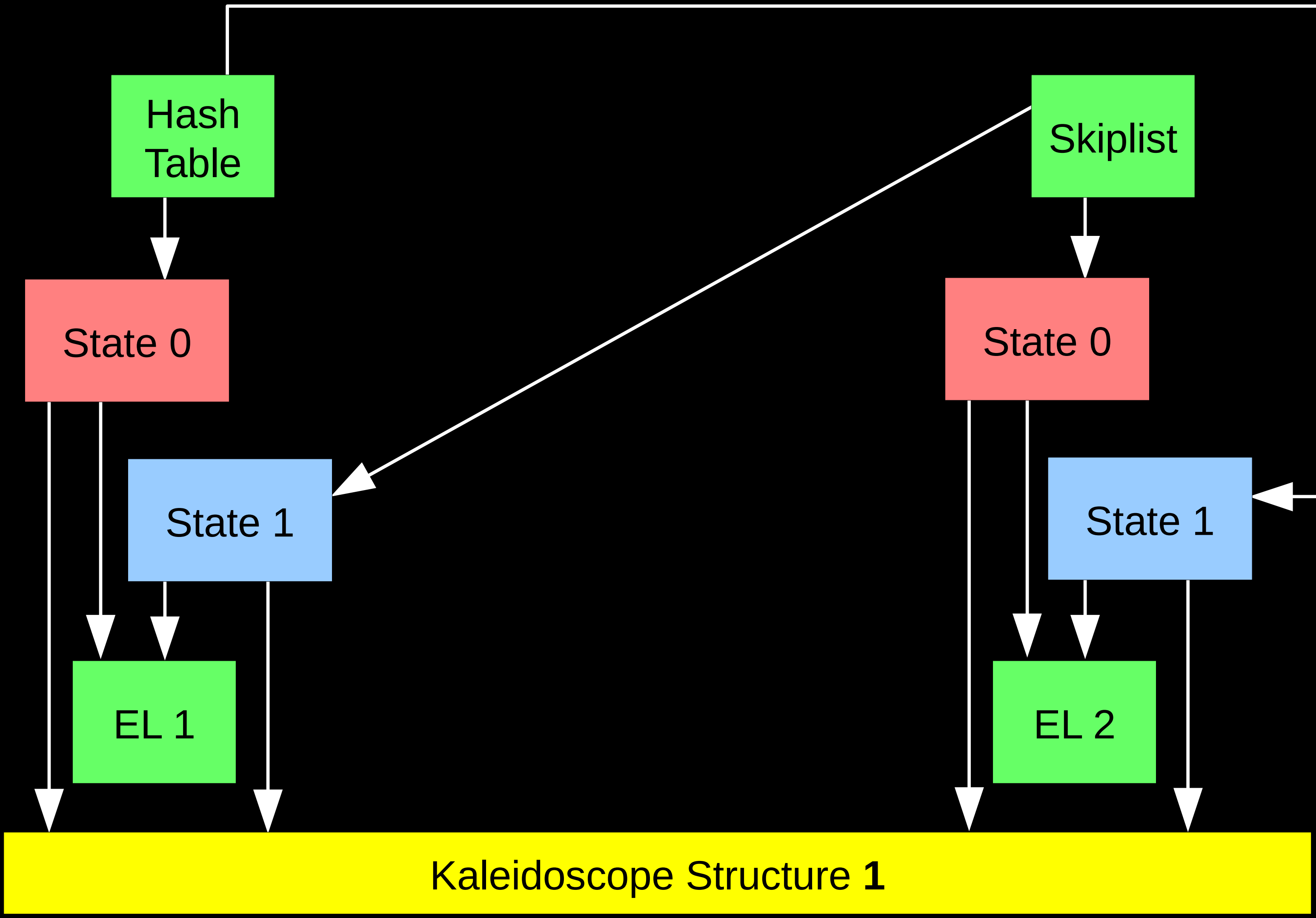
Rotate Through Hash Table & Skiplist (1/3)



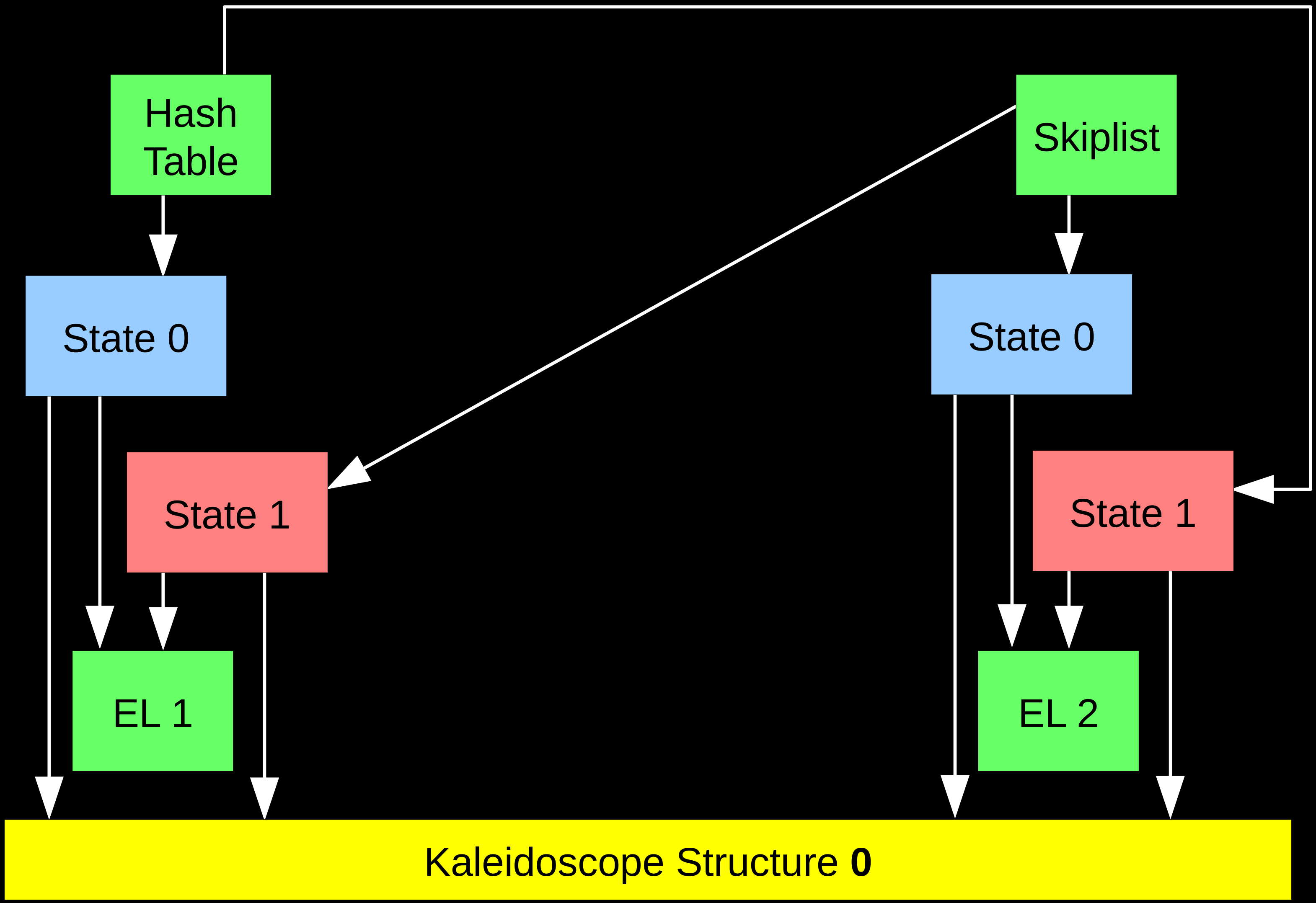
Rotate Through Hash Table & Skiplist (2/3)



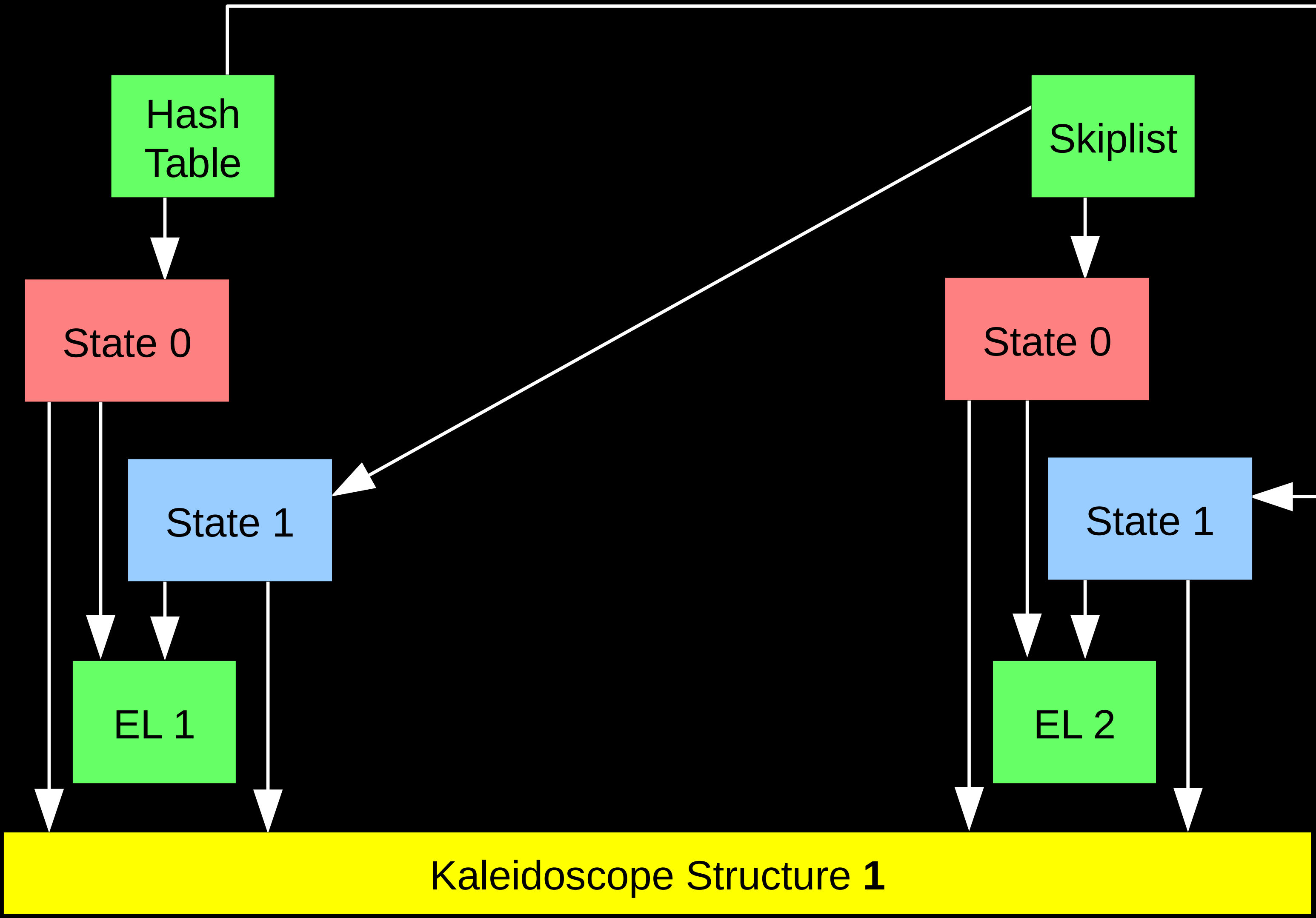
Rotate Through Hash Table & Skiplist (3/3)



Rotate Through Hash Table & Skiplist (2/3)



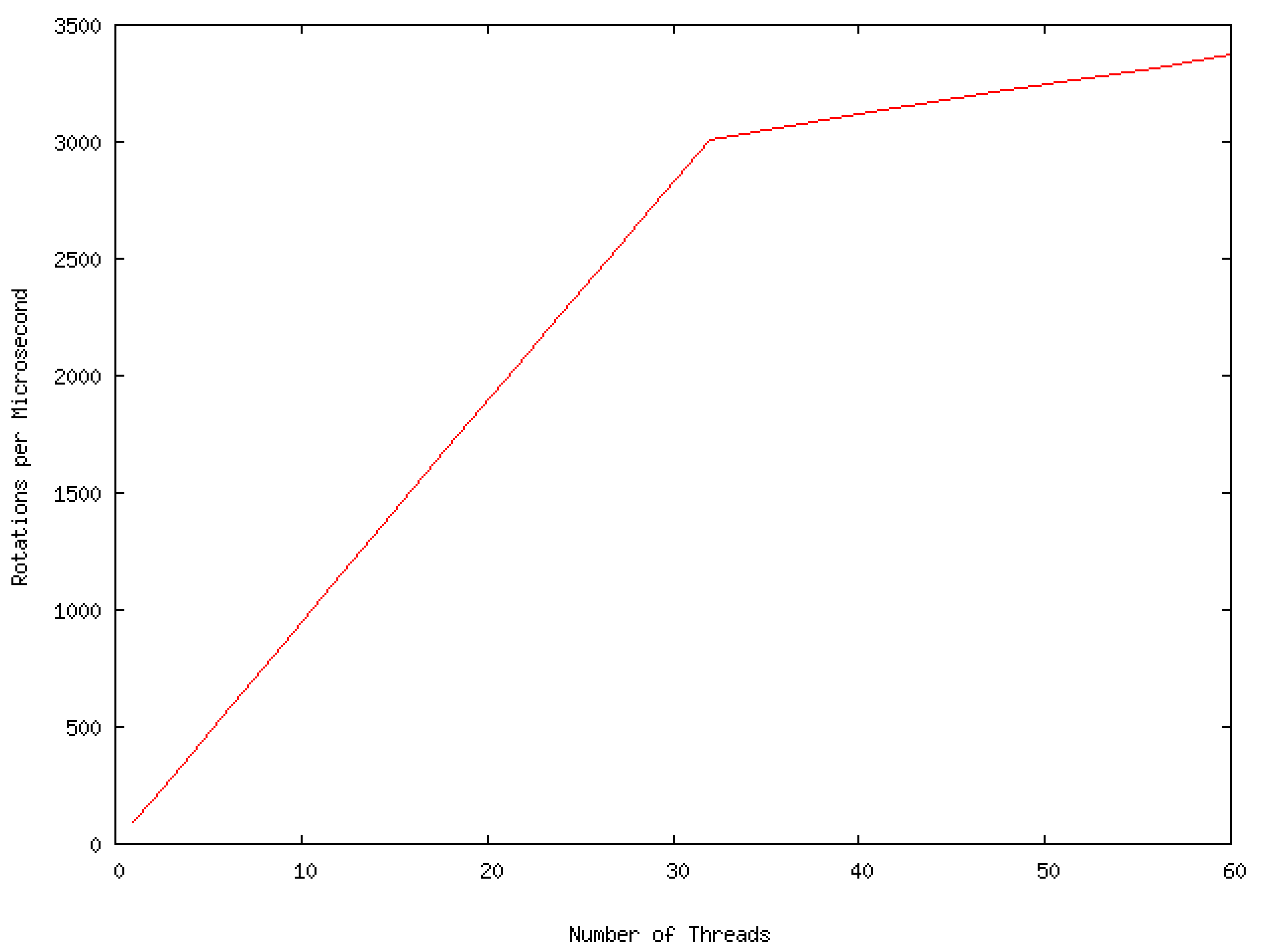
Rotate Through Hash Table & Skiplist (3/3)



Very Tight Loop...

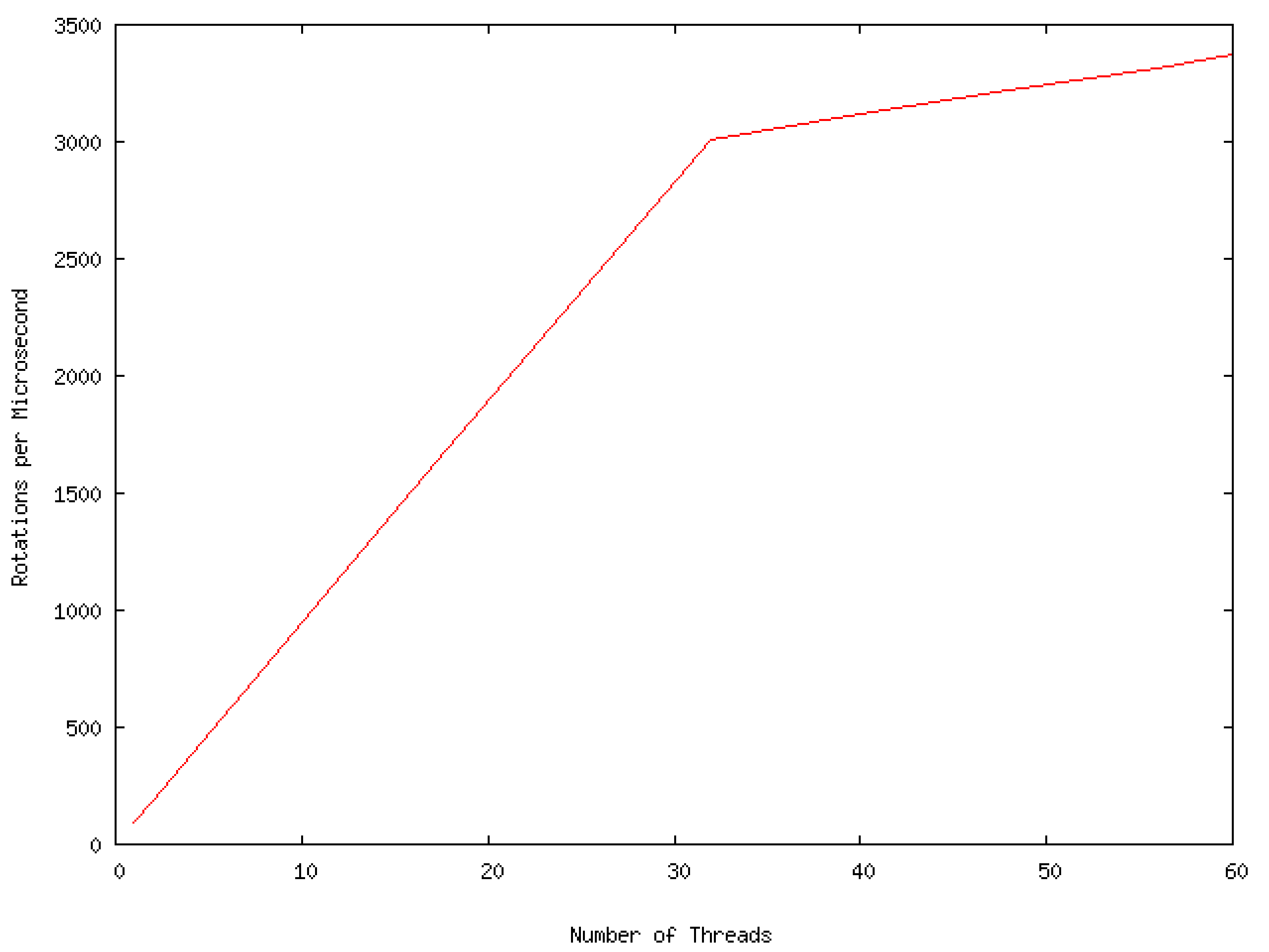
```
while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
        kaleidoscope_set_state(kgp, nrotations % 2);
        nrotations++;
}
```

Kaleidoscopic Rotation Performance Results



This is more like it!!! Too bad about the specificity...

Kaleidoscopic Rotation Performance Results



This is more like it!!! Too bad about the specificity...
As always, be wary of benchmarks!!!

Existence Advantages and Disadvantages

- Existence requires focused developer effort
- Existence specialized to linked structures (for now, anyway)
- Existence requires explicit memory management
- Existence-based exchange operations require linked structures that accommodate duplicate elements
  - Current prototypes disallow duplicates, explicit check for hash tables
- Existence permits irrevocable operations
- Existence can exploit locking hierarchies, reducing the need for contention management
- Existence achieves semi-decent performance and scalability
- Flip/backout automation significantly eases memory management
- Existence's use of synchronization primitives preserves locality of reference
- Existence is compatible with old hardware
- Existence is a downright mean memory-allocator and RCU test case!!!

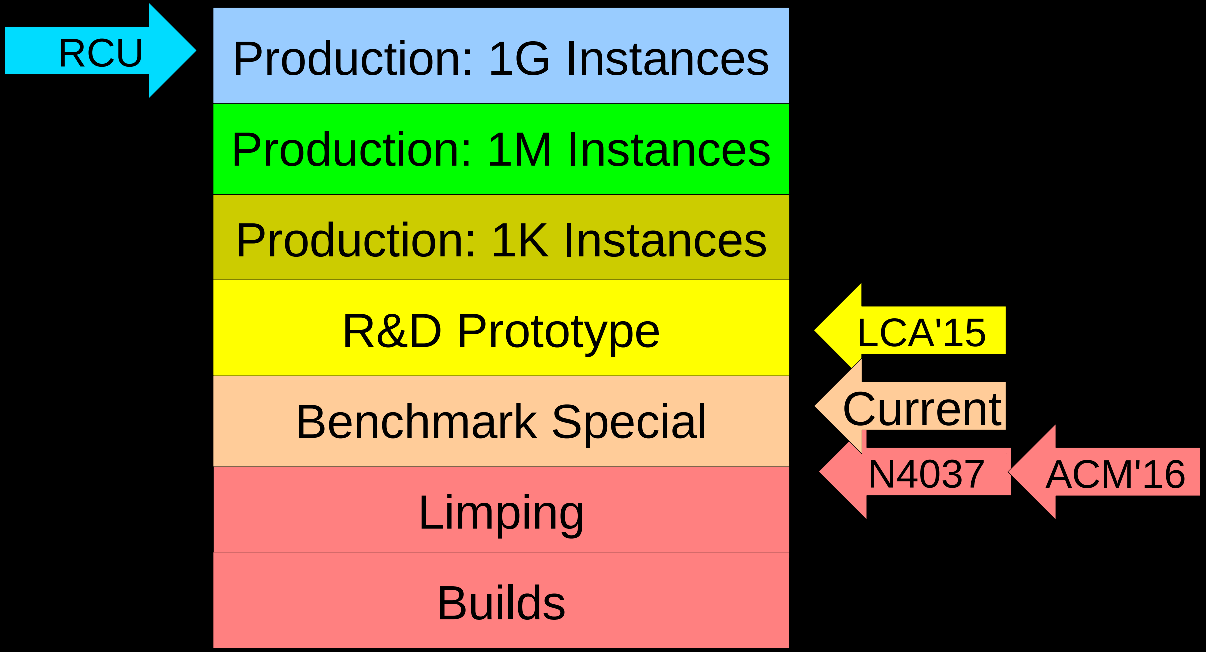
When Might You Use Existence-Based Update?

- We really don't know yet
  - But similar techniques are used by Linux-kernel filesystems
- Best guess is when one or more of the following holds ***and*** you are willing to invest significant developer effort to gain performance and scalability:
  - Many small updates to large linked data structure
  - Complex updates that cannot be efficiently implemented with single pointer update
  - Read-mostly to amortize higher overhead of complex updates
  - Need compatibility with hardware not supporting transactional memory
    - Side benefit: Dispense with the need for software fallbacks!
  - Need to be able to do irrevocable operations (e.g., I/O) as part of data-structure update

Existence Structures: Production Readiness

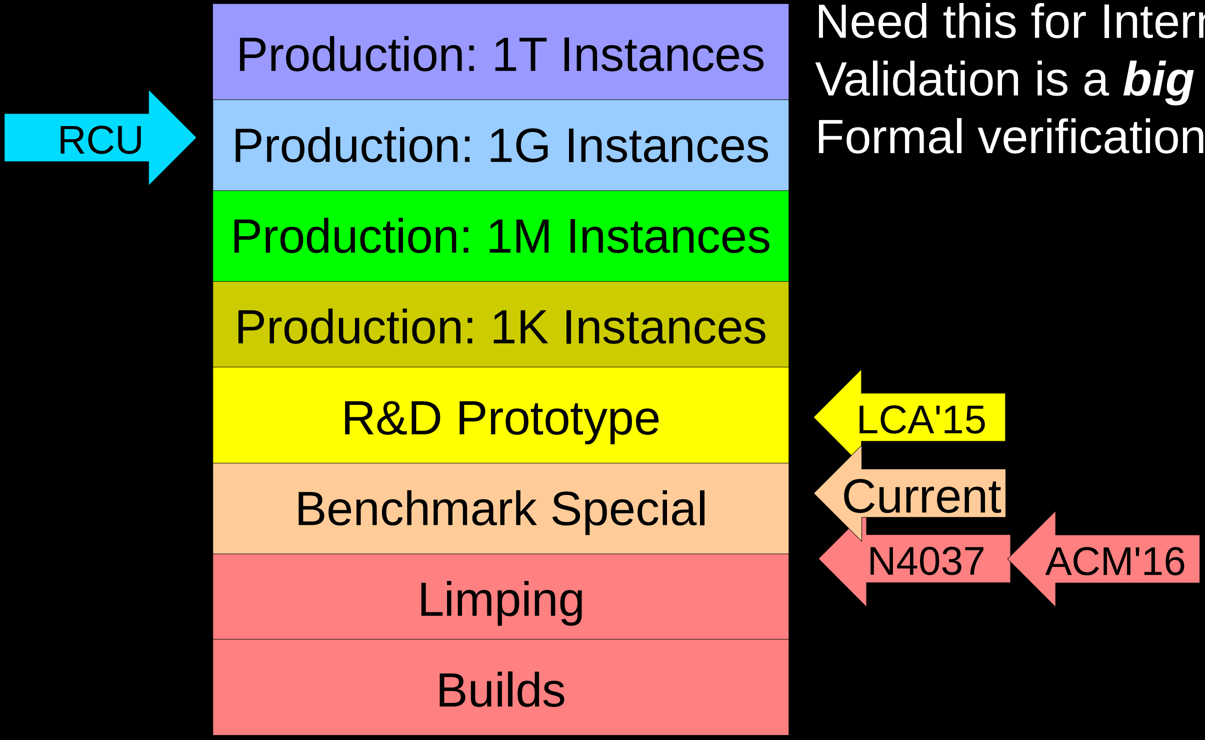
Existence Structures: Production Readiness

- No, it is ***not*** production ready (but was getting there)



Existence Structures: Production Readiness

- No, it is ***not*** production ready (but was getting there)



Need this for Internet of Things,
Validation is a ***big*** unsolved problem
Formal verification for RCU!!!

Existence Structures: Known Antecedents

- Fraser: “Practical Lock-Freedom”, Feb 2004
  - Insistence on lock freedom: High complexity, poor performance
  - Similarity between Fraser's OSTM commit and existence switch
- McKenney, Krieger, Sarma, & Soni: “Atomically Moving List Elements Between Lists Using Read-Copy Update”, Apr 2006
  - Block concurrent operations while large update is carried out
- Triplett: “Scalable concurrent hash tables via relativistic programming”, Sept 2009
- Triplett: “Relativistic Causal Ordering: A Memory Model for Scalable Concurrent Data Structures”, Feb 2012
  - Similarity between Triplett's key switch and allegiance switch
  - Could share nodes between trees like Triplett does between hash chains, but would impose restrictions and API complexity
- Some filesystem algorithms in Linux kernel

Summary

Summary

- Complex atomic updates can be applied to unmodified RCU-aware concurrent data structures
  - Need functions to add, remove, and free elements
  - Free to use any synchronization mechanism
  - Free to use any memory allocator
- Flip/backout processing can be automated
- High update rates encounter interesting bottlenecks in the infrastructure: Memory allocation and userspace RCU
  - Read-mostly workloads continue to perform and scale well
  - As do kaleidoscopic updates
- Lots of opportunity for collaboration and innovation!

Graphical Summary

Z
Z
Z
z
z
RCU
Hazard Pointers

Not only are they lazy, they get more work done than I do!
'yawn'
TM
Locking

To Probe Deeper (1/4)

- Hash tables:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Chapter 10
- Split counters:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Chapter 5
  - http://events.linuxfoundation.org/sites/events/files/slides/BareMetal.2014.03.09a.pdf
- Perfect partitioning
  - Candide et al: “Dynamo: Amazon's highly available key-value store”
    - http://doi.acm.org/10.1145/1323293.1294281
  - McKenney: “Is Parallel Programming Hard, And, If So, What Can You Do About It?”
    - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 6.5
  - McKenney: “Retrofitted Parallelism Considered Grossly Suboptimal”
    - Embarrassing parallelism vs. humiliating parallelism
    - https://www.usenix.org/conference/hotpar12/retro%EF%AC%81tted-parallelism-considered-grossly-sub-optimal
  - McKenney et al: “Experience With an Efficient Parallel Kernel Memory Allocator”
    - http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf
  - Bonwick et al: “Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”
    - http://static.usenix.org/event/usenix01/full_papers/bonwick/bonwick_html/
  - Turner et al: “PerCPU Atomics”
    - http://www.linuxplumbersconf.org/2013/ocw//system/presentations/1695/original/LPC%20-%20PerCpu%20Atomics.pdf

To Probe Deeper (2/4)

- Stream-based applications:
  - Sutton: “Concurrent Programming With The Disruptor”
    - http://www.youtube.com/watch?v=UvE389P6Er4
    - http://lca2013.linux.org.au/schedule/30168/view_talk
  - Thompson: “Mechanical Sympathy”
    - http://mechanical-sympathy.blogspot.com/
- Read-only traversal to update location
  - Arcangeli et al: “Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel”
    - https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html
  - Corbet: “Dcache scalability and RCU-walk”
    - https://lwn.net/Articles/419811/
  - Xu: “bridge: Add core IGMP snooping support”
    - http://kerneltrap.com/mailarchive/linux-netdev/2010/2/26/6270589
  - Triplett et al., “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
    - http://www.usenix.org/event/atc11/tech/final_files/Triplett.pdf
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
    - http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf
  - McKenney et al: “URCU-Protected Hash Tables”
    - http://lwn.net/Articles/573431/

To Probe Deeper (3/4)

- Hardware lock elision: Overviews
  - Kleen: “Scaling Existing Lock-based Applications with Lock Elision”
    - http://queue.acm.org/detail.cfm?id=2579227
- Hardware lock elision: Hardware description
  - POWER ISA Version 2.07
    - http://www.power.org/documentation/power-isa-version-2-07/
  - Intel® 64 and IA-32 Architectures Software Developer Manuals
    - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
  - Jacobi et al: “Transactional Memory Architecture and Implementation for IBM System z”
    - http://www.microsymposia.org/micro45/talks-posters/3-jacobi-presentation.pdf
- Hardware lock elision: Evaluations
  - http://pcl.intel-research.net/publications/SC13-TSX.pdf
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 16.3
- Hardware lock elision: Need for weak atomicity
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
    - http://research.sun.com/scalable/pubs/PODC03.pdf
  - Shavit et al: “Data structures in the multicore age”
    - http://doi.acm.org/10.1145/1897852.1897873
  - Haas et al: “How FIFO is your FIFO queue?”
    - http://dl.acm.org/citation.cfm?id=2414731
  - Gramoli et al: “Democratizing transactional programming”
    - http://doi.acm.org/10.1145/2541883.2541900

To Probe Deeper (4/4)

- RCU
  - Desnoyers et al.: “User-Level Implementations of Read-Copy Update”
    - http://www.rdrop.com/users/paulmck/RCU/urcu-main-accepted.2011.08.30a.pdf
    - http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf
  - McKenney et al.: “RCU Usage In the Linux Kernel: One Decade Later”
    - http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf
    - http://rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf
  - McKenney: “Structured deferral: synchronization via procrastination”
    - http://doi.acm.org/10.1145/2483852.2483867
  - McKenney et al.: “User-space RCU” https://lwn.net/Articles/573424/
- Possible future additions
  - Boyd-Wickizer: “Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels”
    - http://pdos.csail.mit.edu/papers/sbw-phd-thesis.pdf
  - Clements et al: “The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors”
    - http://www.read.seas.harvard.edu/~kohler/pubs/clements13scalable.pdf
  - McKenney: “N4037: Non-Transactional Implementation of Atomic Tree Move”
    - http://www.rdrop.com/users/paulmck/scalability/paper/AtomicTreeMove.2014.05.26a.pdf
  - McKenney: “C++ Memory Model Meets High-Update-Rate Data Structures”
    - http://www2.rdrop.com/users/paulmck/RCU/C++Updates.2014.09.11a.pdf

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- Credits:
  - This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Mathieu Desnoyers, Alan Stern, Michel Dagenais, Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, Jonathan Walpole, and the Linux kernel community.
  - Additional reviewers: Carsten Weinhold and Mingming Cao.

Questions?

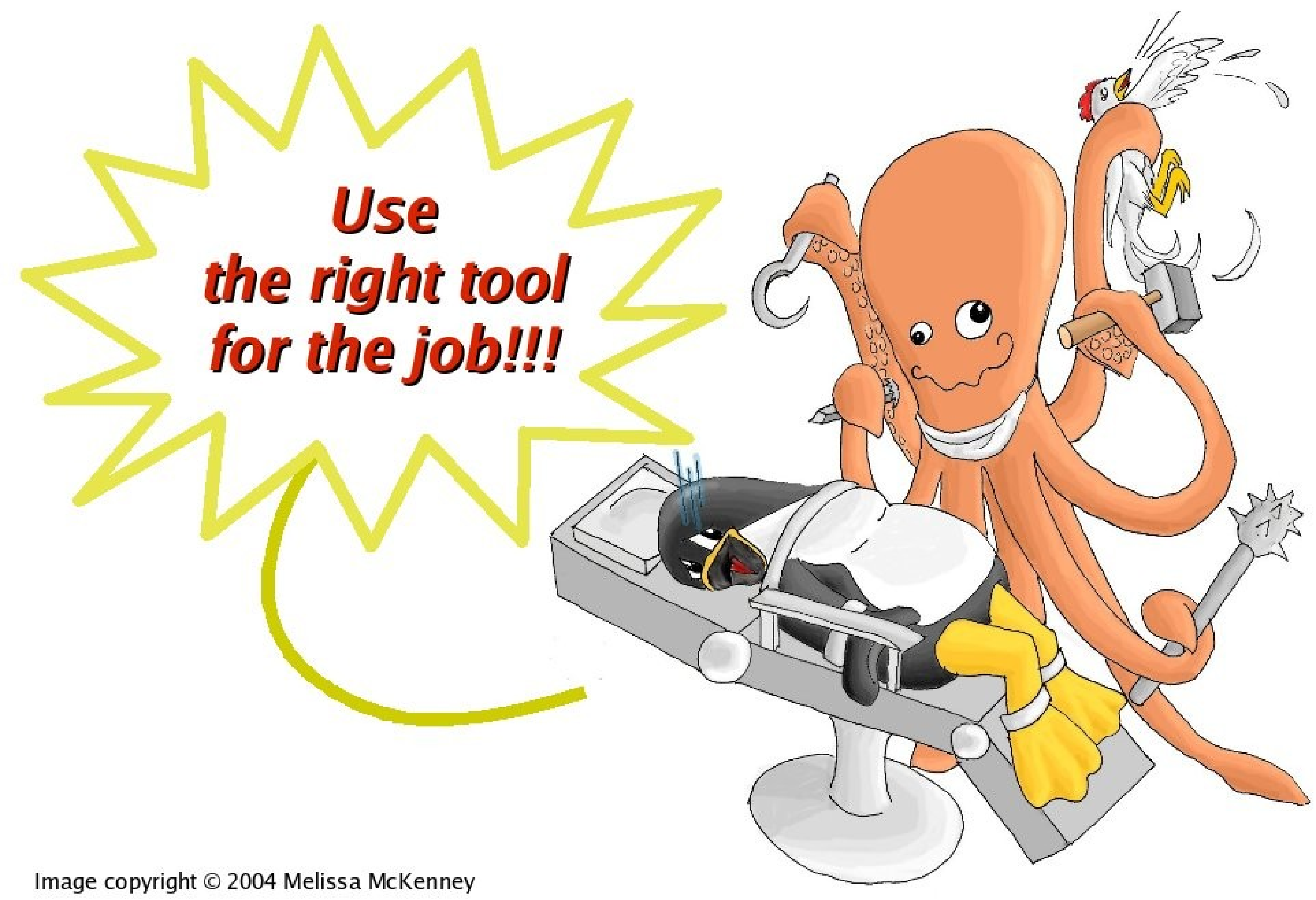


Image copyright © 2004 Melissa McKenney