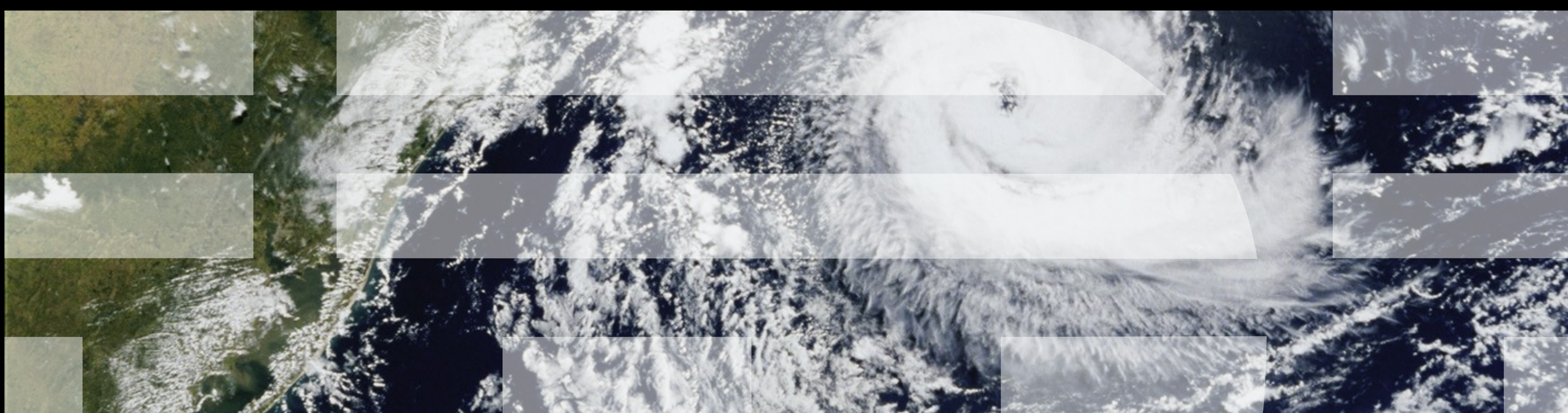
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
SVLUG, September 11, 2017

Beyond the Issaquah Challenge:

High-Performance Scalable Complex Updates

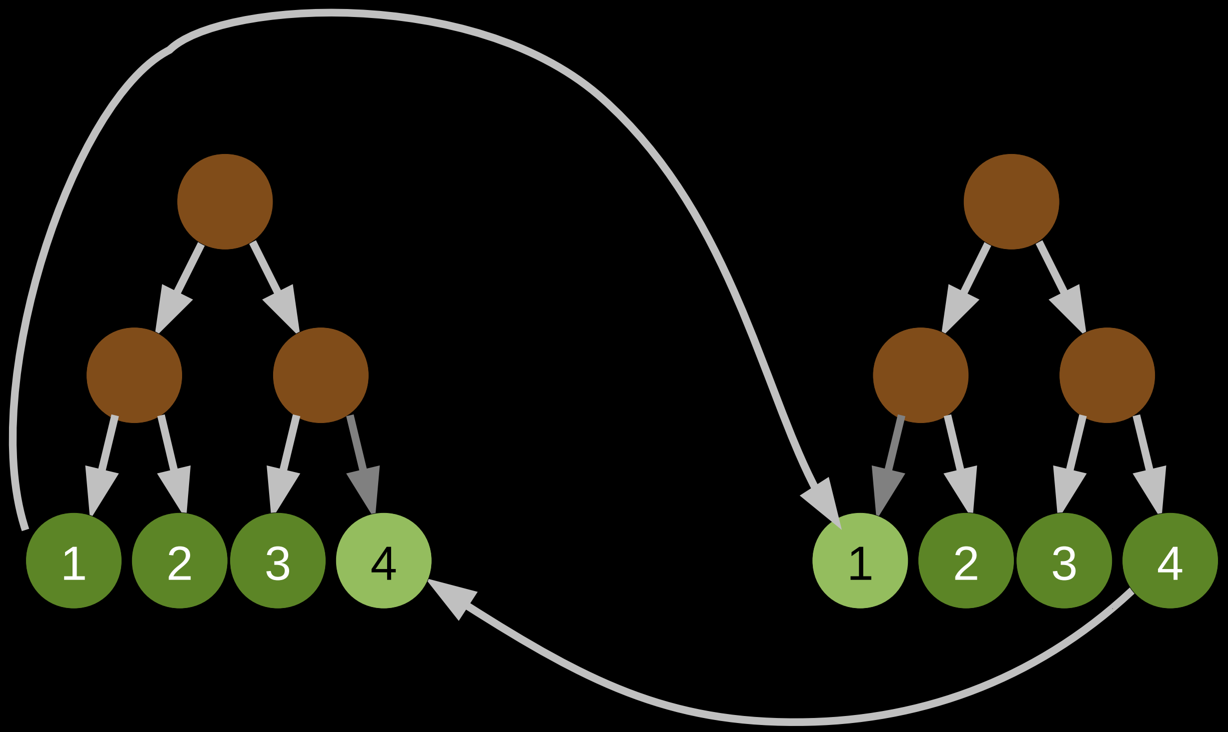


Overview

- The Issaquah Challenge
- Parallelism and the Laws of Physics
- Special Case for Parallel Updates
- The Issaquah Challenge: Complex-Update Solutions

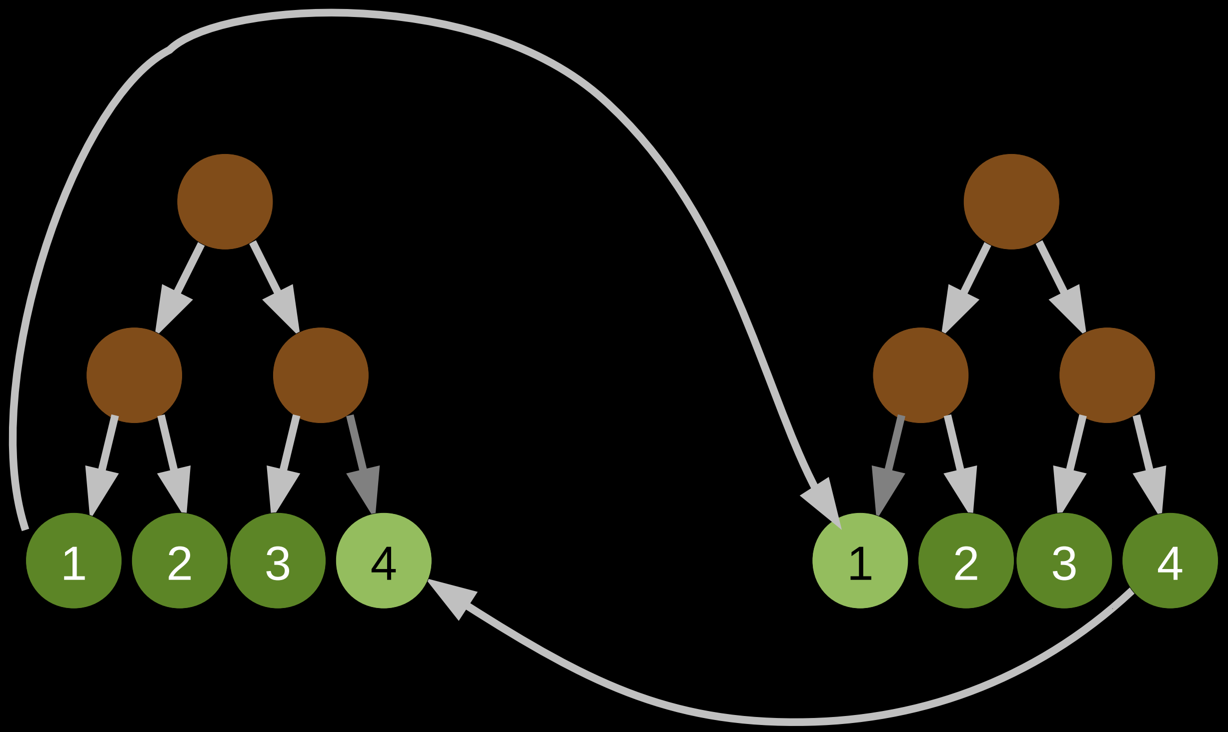
Atomic Multi-Structure Update: Issaquah Challenge

Atomic Multi-Structure Update: Issaquah Challenge



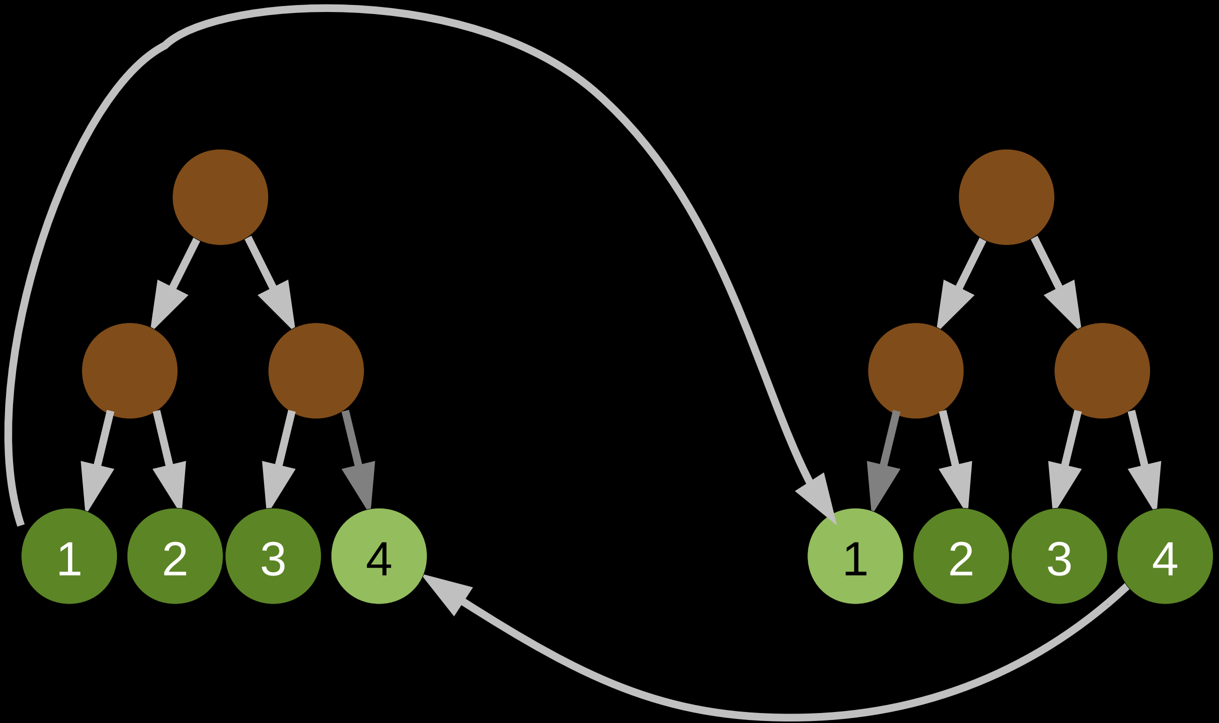
Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree

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



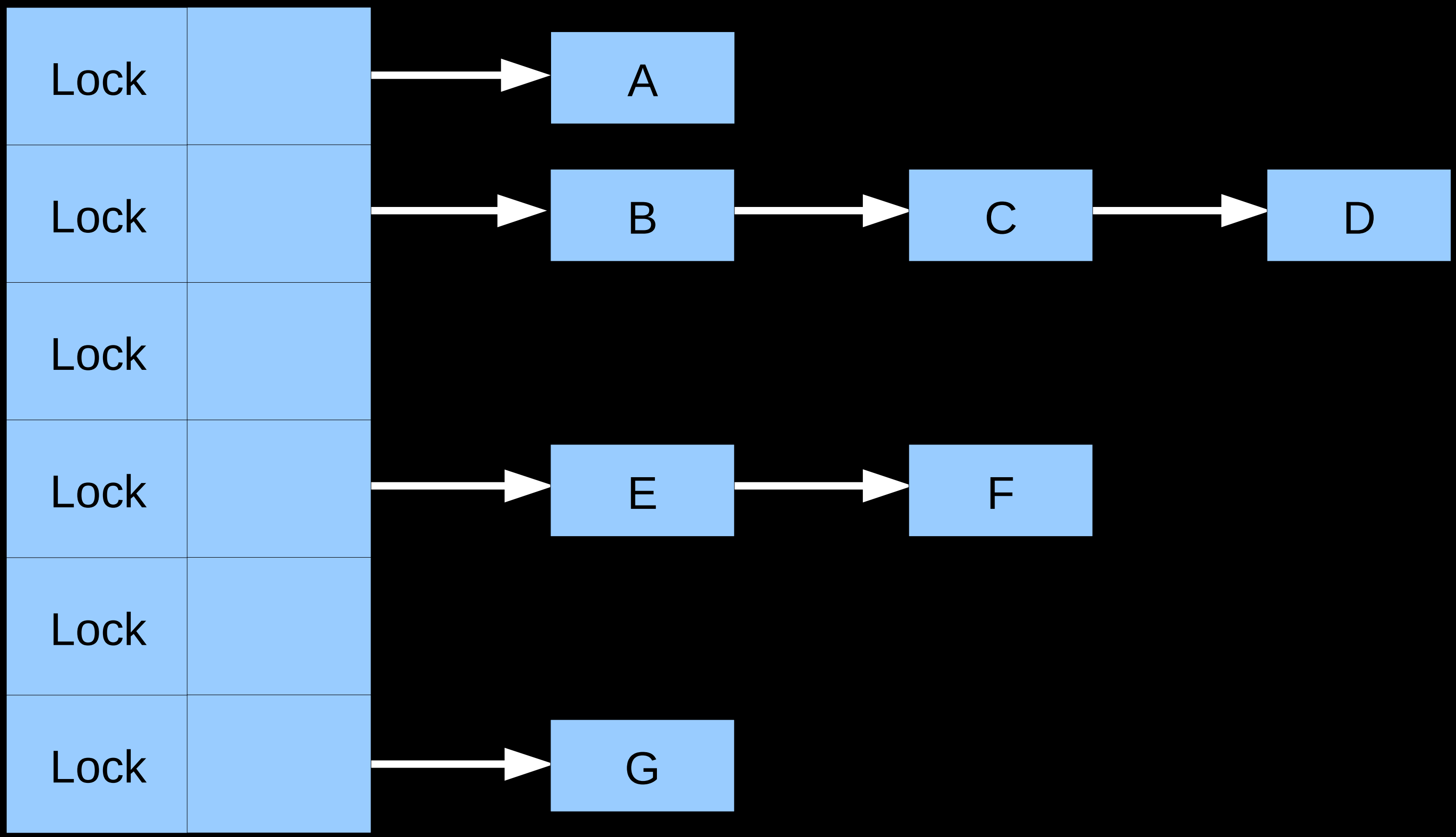
Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree
Without contention between the two move operations!
Hence, most locking solutions “need not apply”

Issaquah Update: History

- N4037 (May 2014): Crude first solution
- CPPCON (September 2014): Some scalability
- LCA (January 2015): Decent scalability, minor modifications to RCU algorithms for complex atomic updates, OK reliability
- ACM Applicative Conference (June 2016):
  - Fewer levels of indirection, courtesy of Dmitry Vyukov
  - Wrappering: RCU-enabled concurrent data structures used unchanged
  - Automated cleanup after atomic update or backout
  - Improved reliability
  - But starting from ground zero on scalability and reliability!
- CPPCON (September 2016)
  - Updates involving multiple types of data structures
- SVLUG (September 2017)
  - You are here!!!

But Aren't Parallel Updates A Solved Problem?

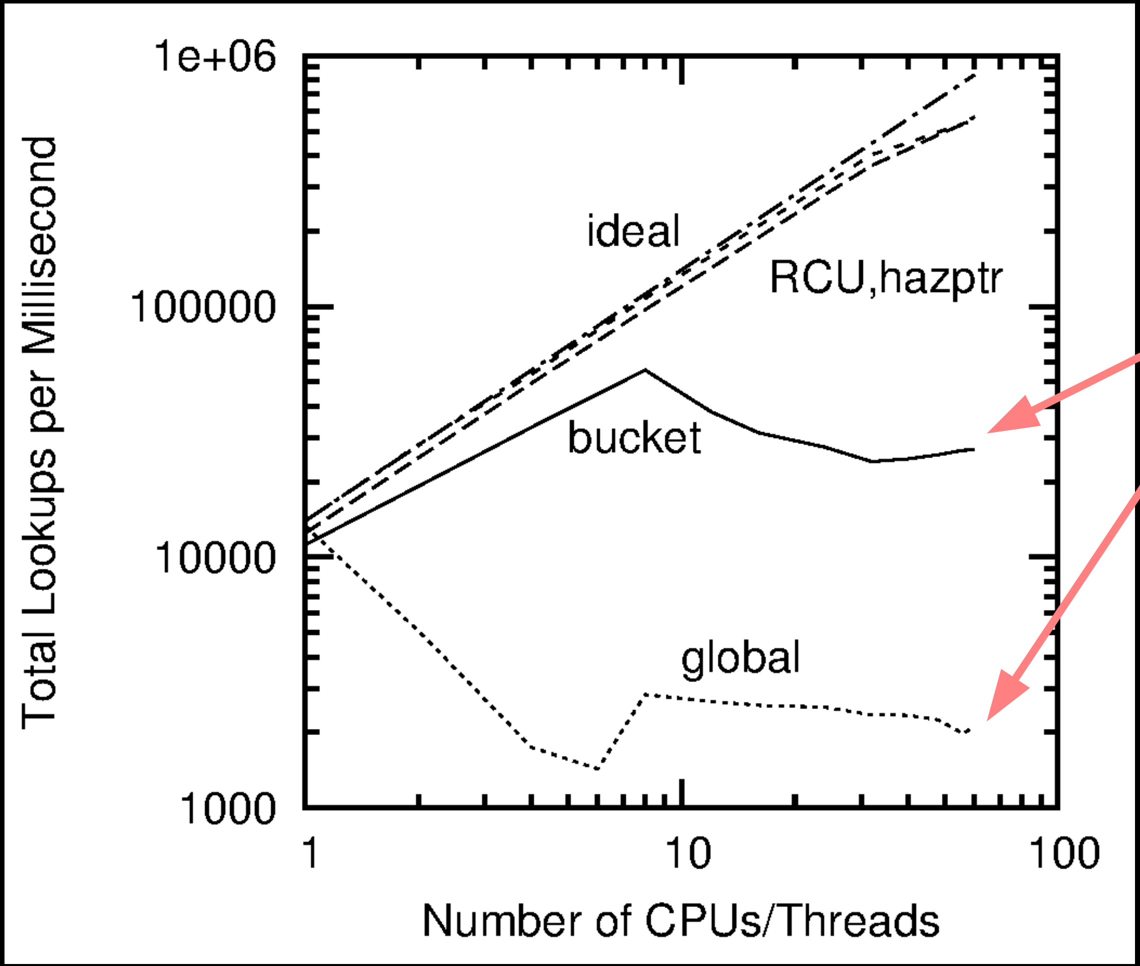
Parallel-Processing Workhorse: Hash Tables



Perfect partitioning leads to perfect performance and stunning scalability!

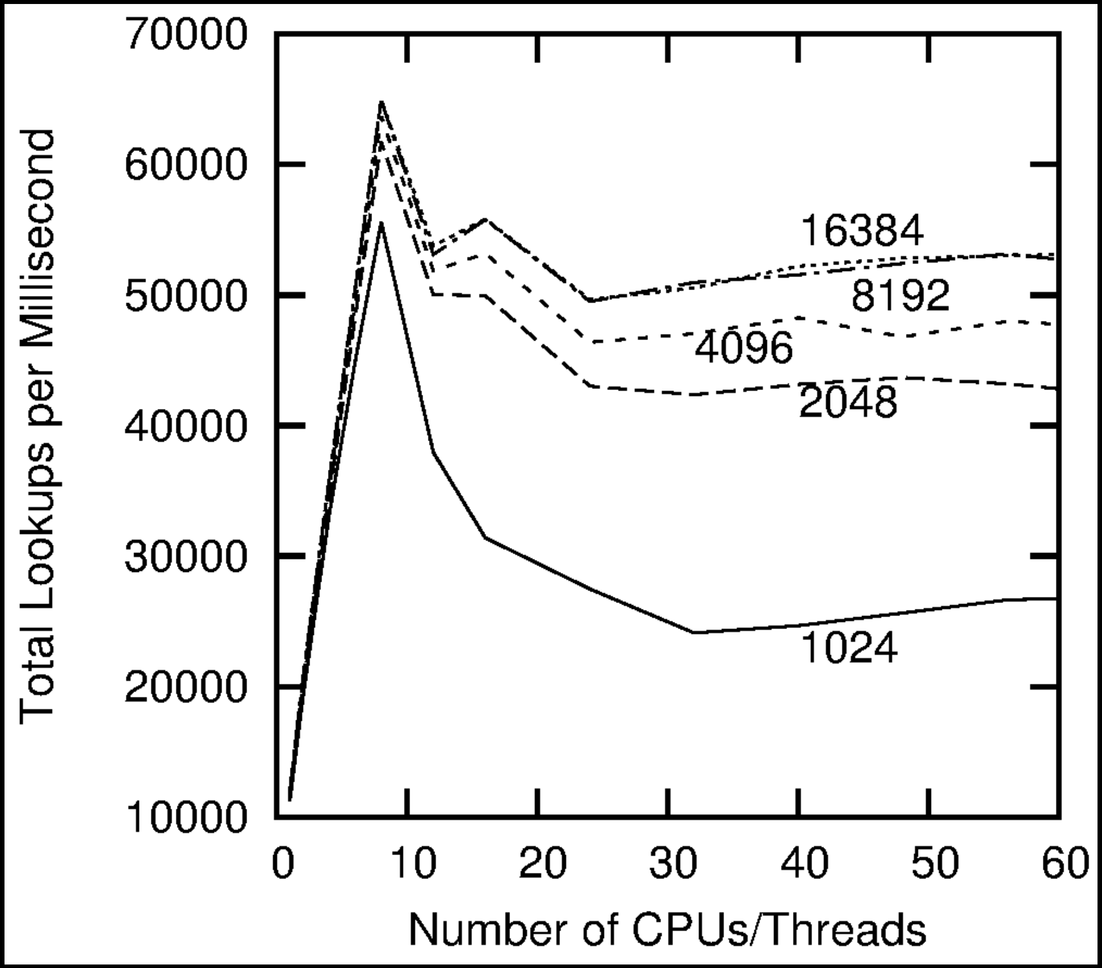
In theory, anyway...

Read-Mostly Workloads Scale Very Nicely, Update-Heavy Workloads, Not So Much...



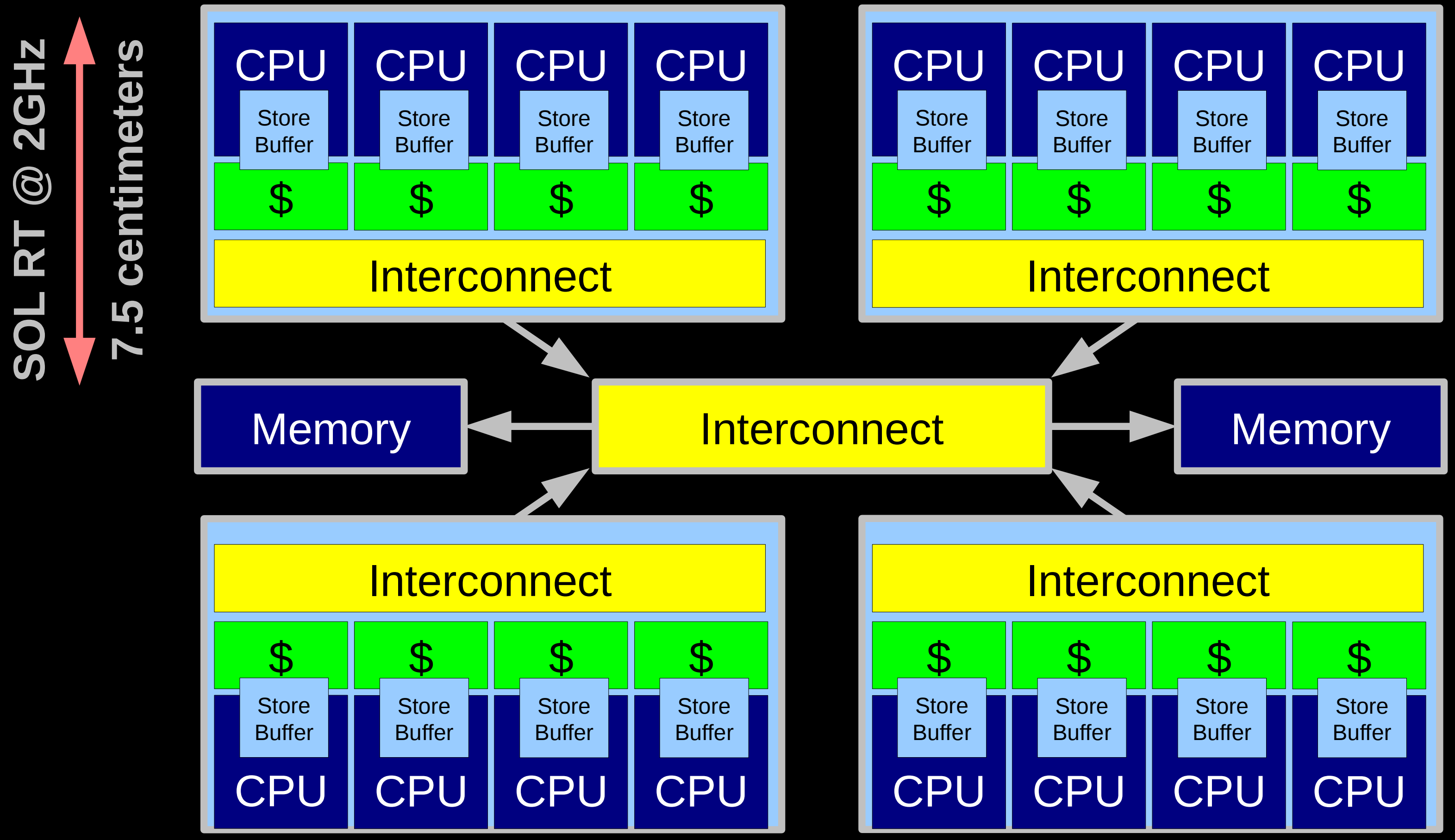
And the horrible thing? Updates are all locking ops!

But Hash Tables Are Partitionable! # of Buckets?



Some improvement, but...

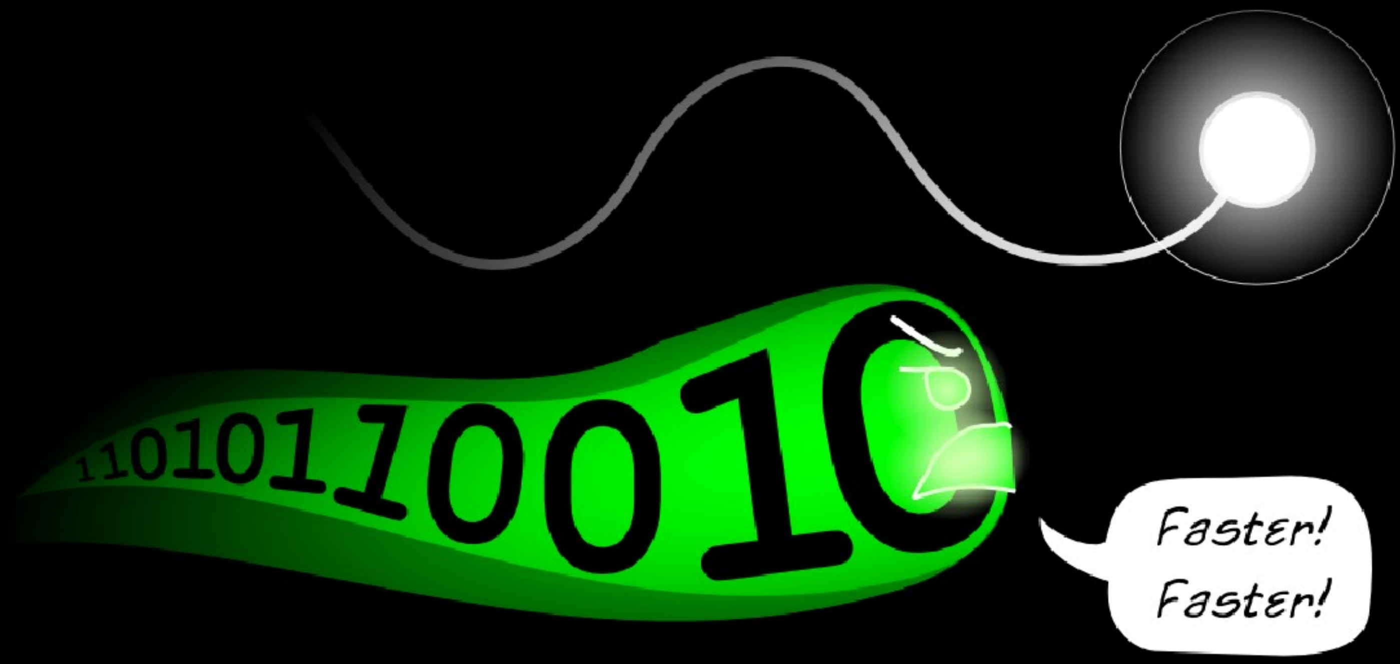
Hardware Structure and Laws of Physics



Electrons move at 0.03C to 0.3C in transistors and, so need locality of reference

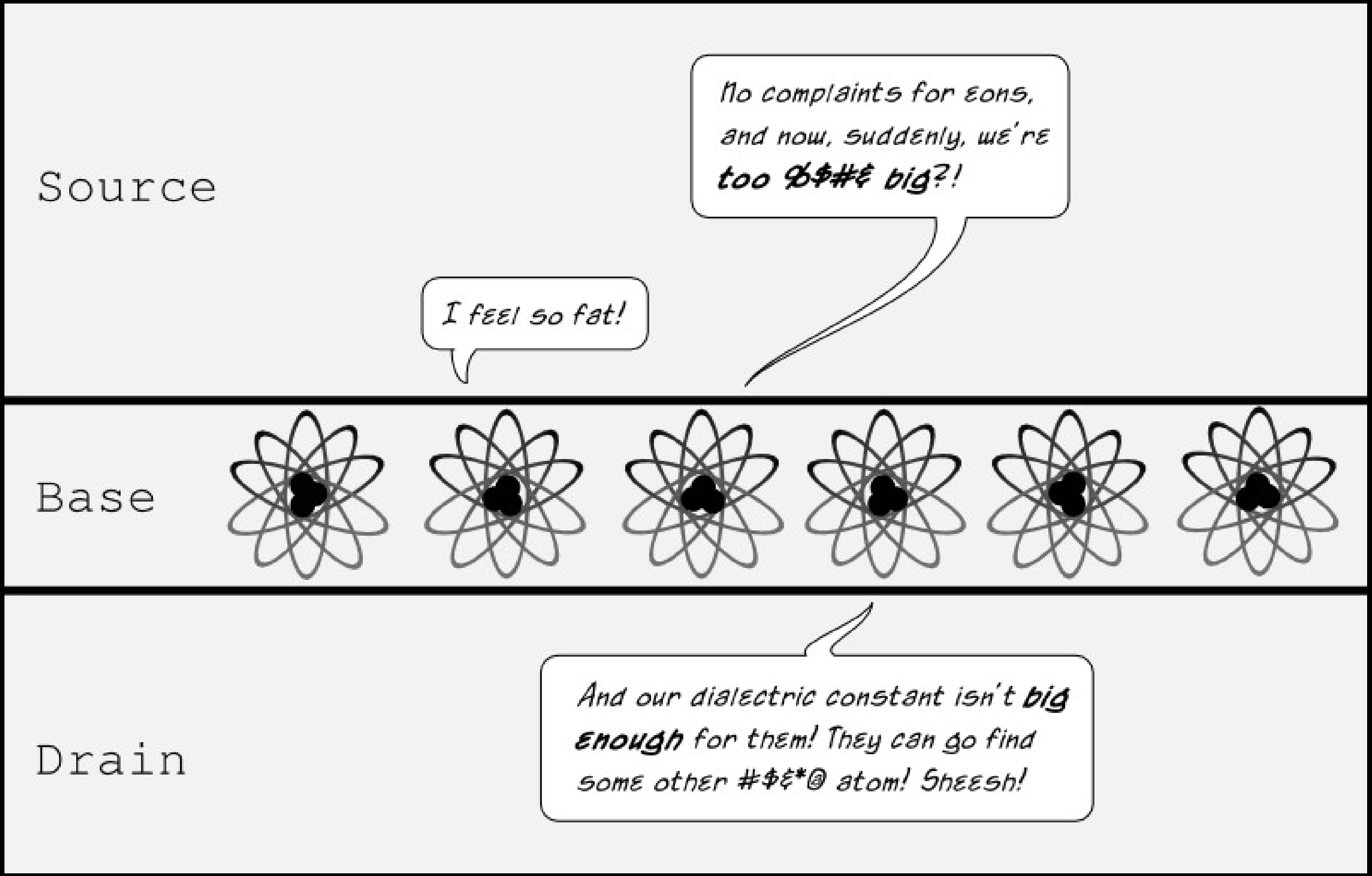
Two Problems With Fundamental Physics...

Problem With Physics #1: Finite Speed of Light



Observation by Stephen Hawking

Problem With Physics #2: Atomic Nature of Matter

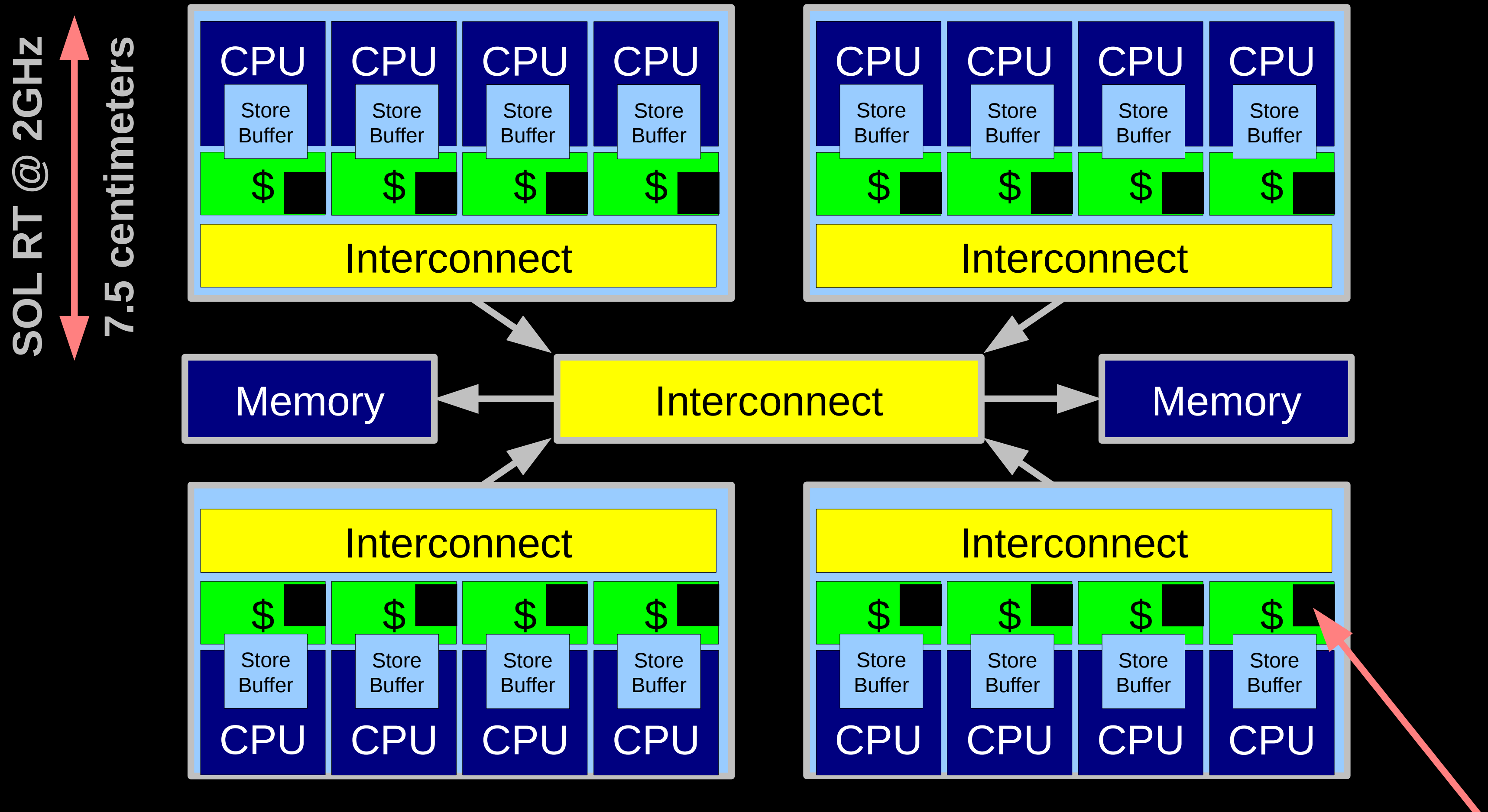


Observation by Stephen Hawking

We Do Not Yet Know How to Abstract Away the Laws of Physics

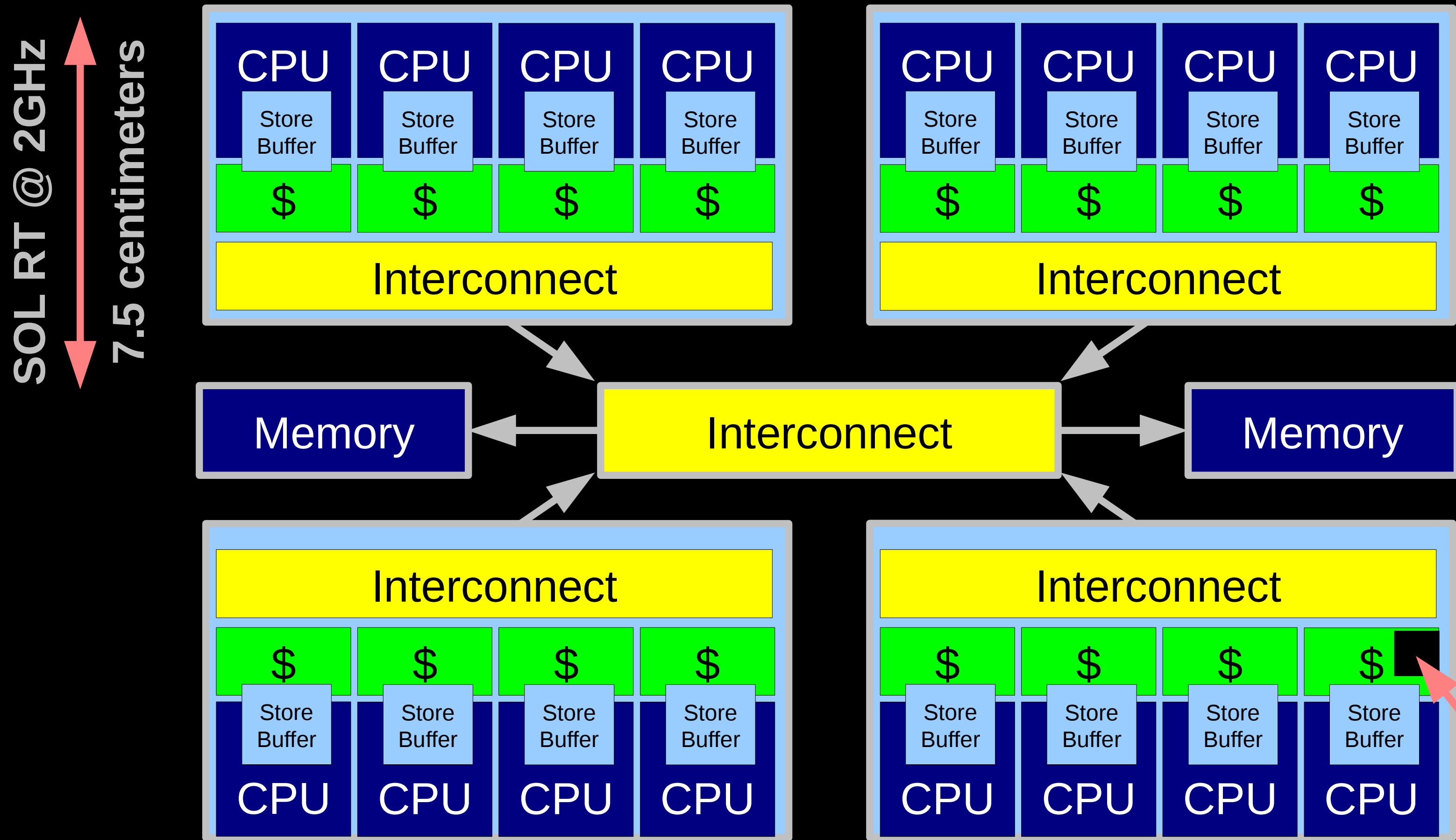
However, There Are Ways to Work Around Them!

Read-Mostly Access Dodges The Laws of Physics!!!



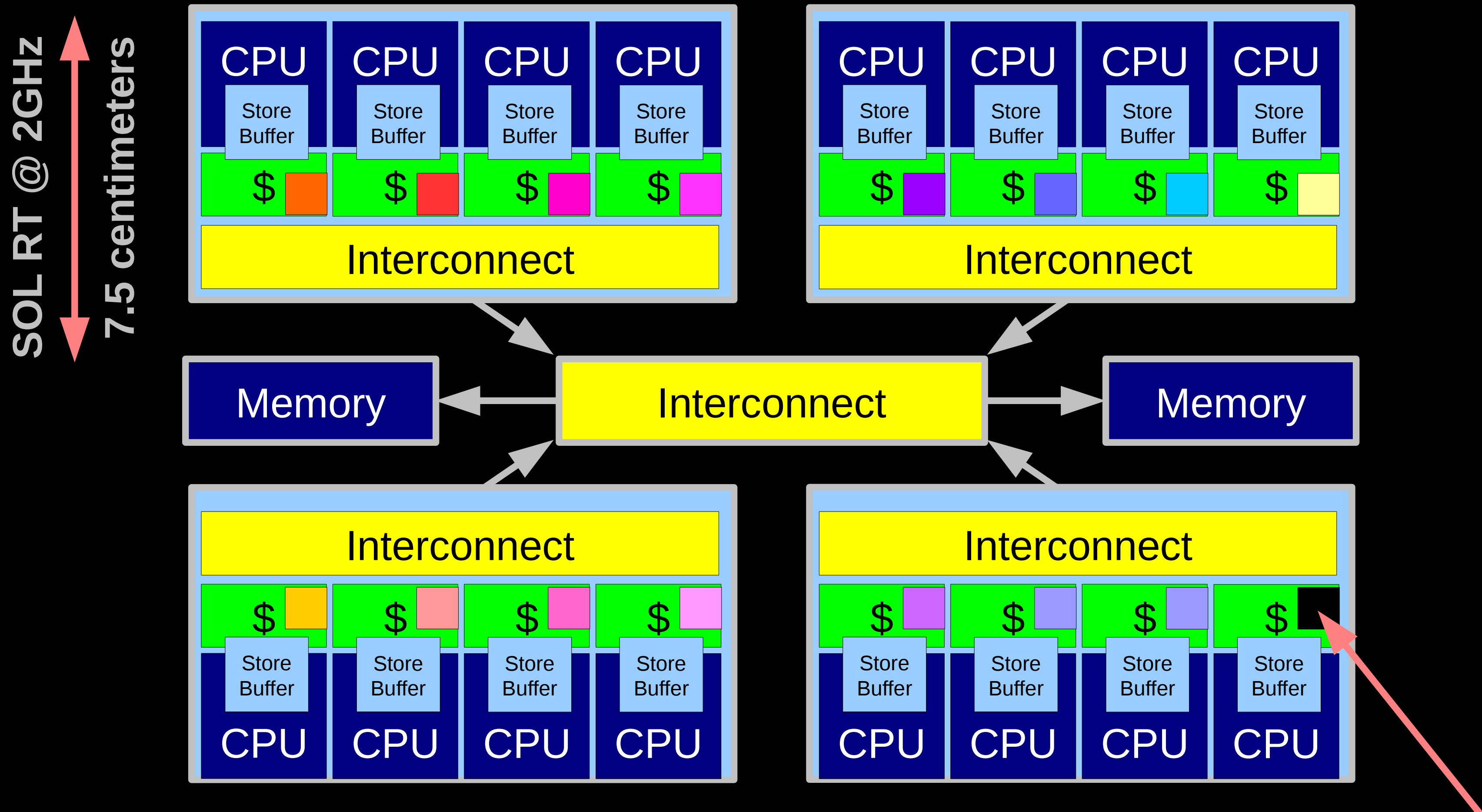
Read-only data remains replicated in all caches

Updates, Not So Much...



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

Updates, Not So Much... Must Leverage Locality!



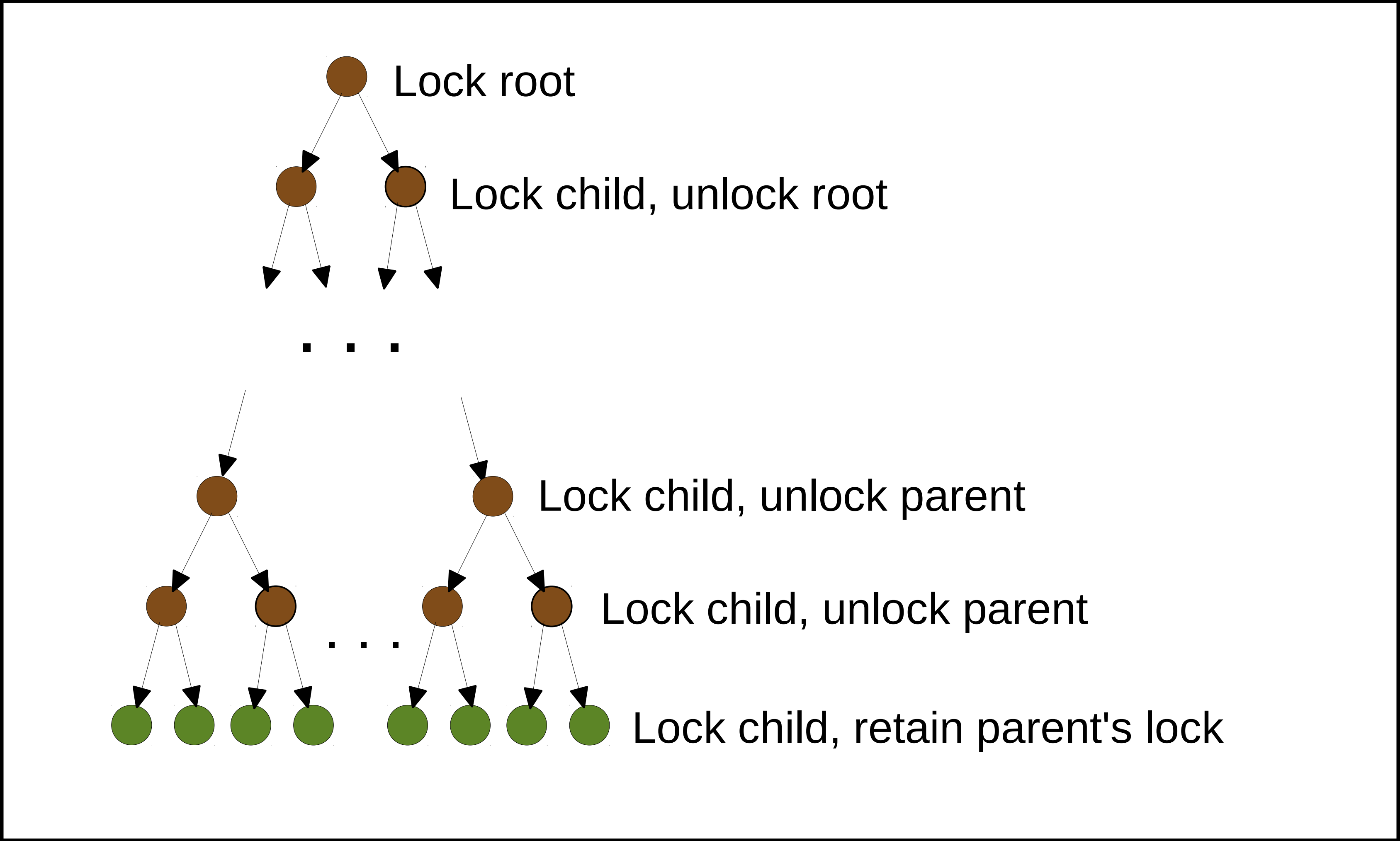
Each CPU operates on its own "shard" of the data, preserving cache locality and performance

Dodging the Laws of Physics for Updates

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

Read-Only Traversal To Location Being Updated

Why Read-Only Traversal To Update Location?



Lock contention despite read-only accesses!

And This Is Why We Have RCU!

- Lightest-weight conceivable read-side primitives

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

- Advantages:

- Disadvantage:

Quick overview, references at end of slideset

And This Is Why We Have RCU!

- Lightest-weight conceivable read-side primitives

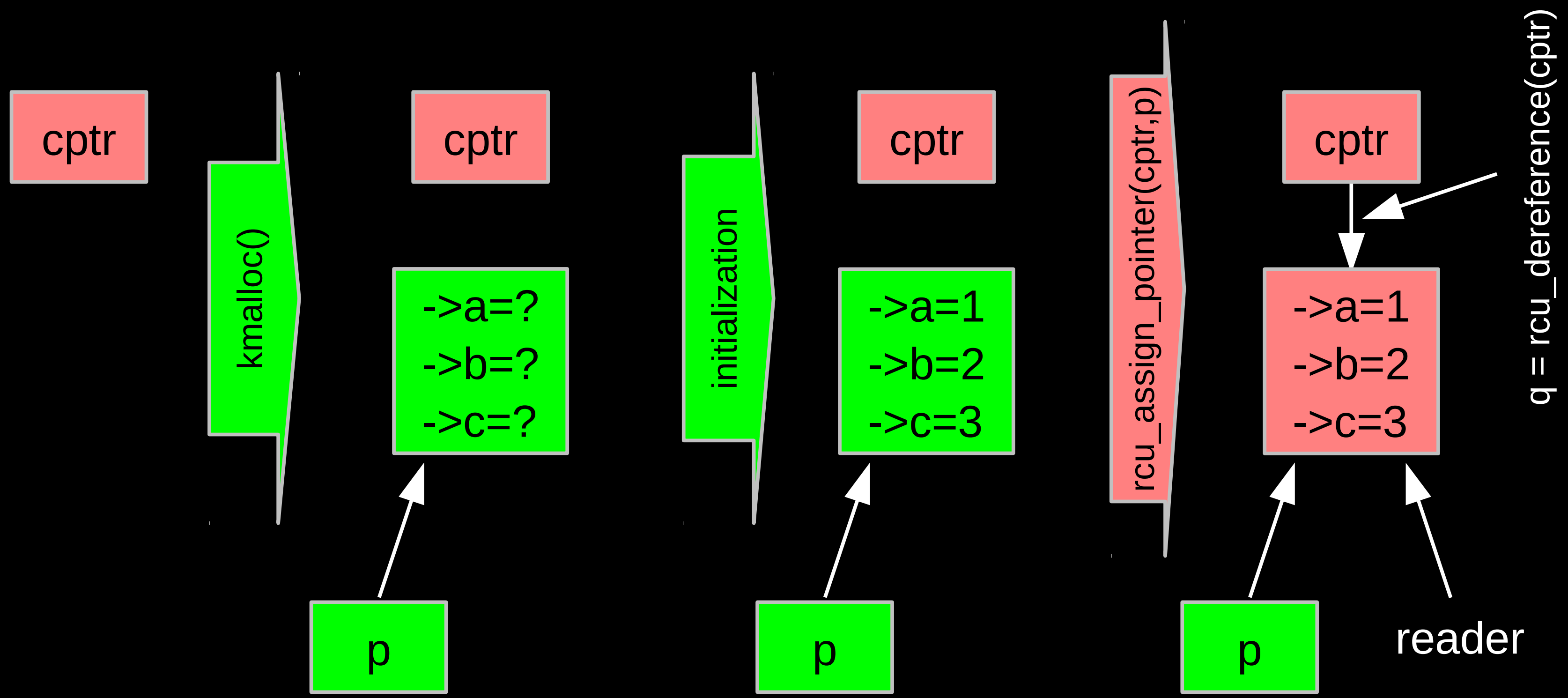
```
/* Assume non-preemptible (run-to-block) environment. */
#define rcu_read_lock()
#define rcu_read_unlock()
```

- 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???

Quick overview, references at end of slideset

RCU Addition to a Linked Structure

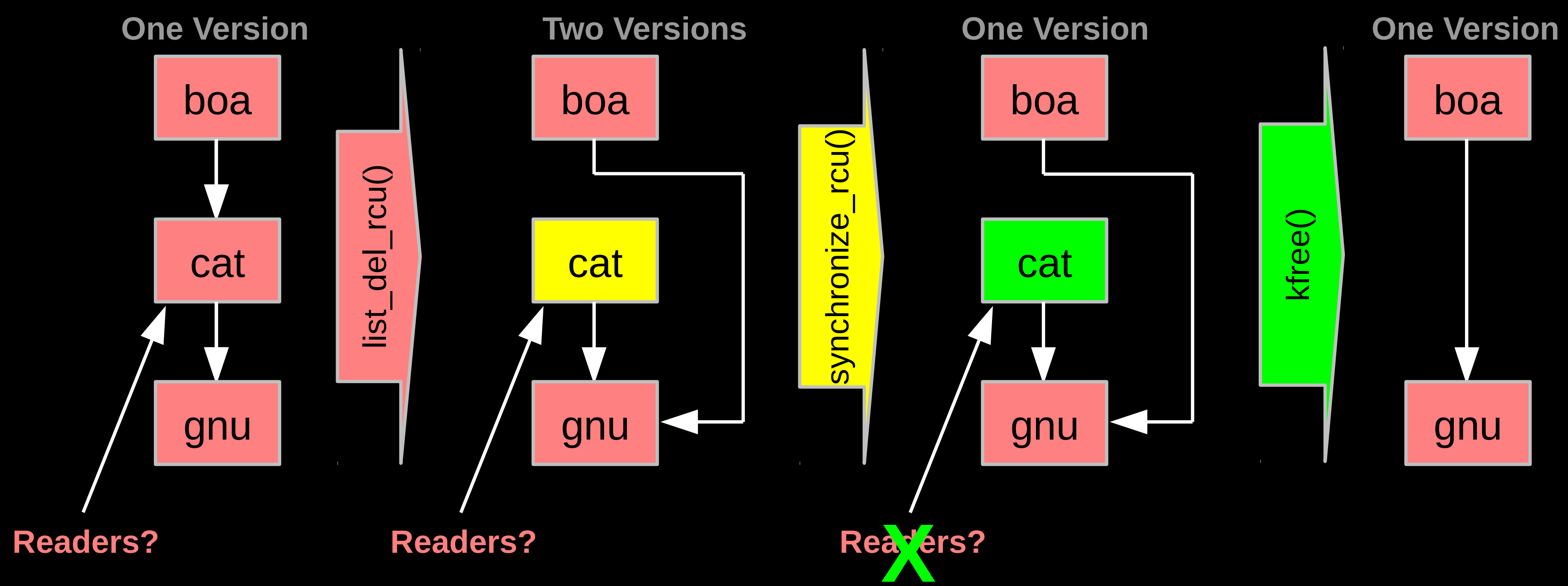
Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers



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

RCU Safe Removal From Linked Structure

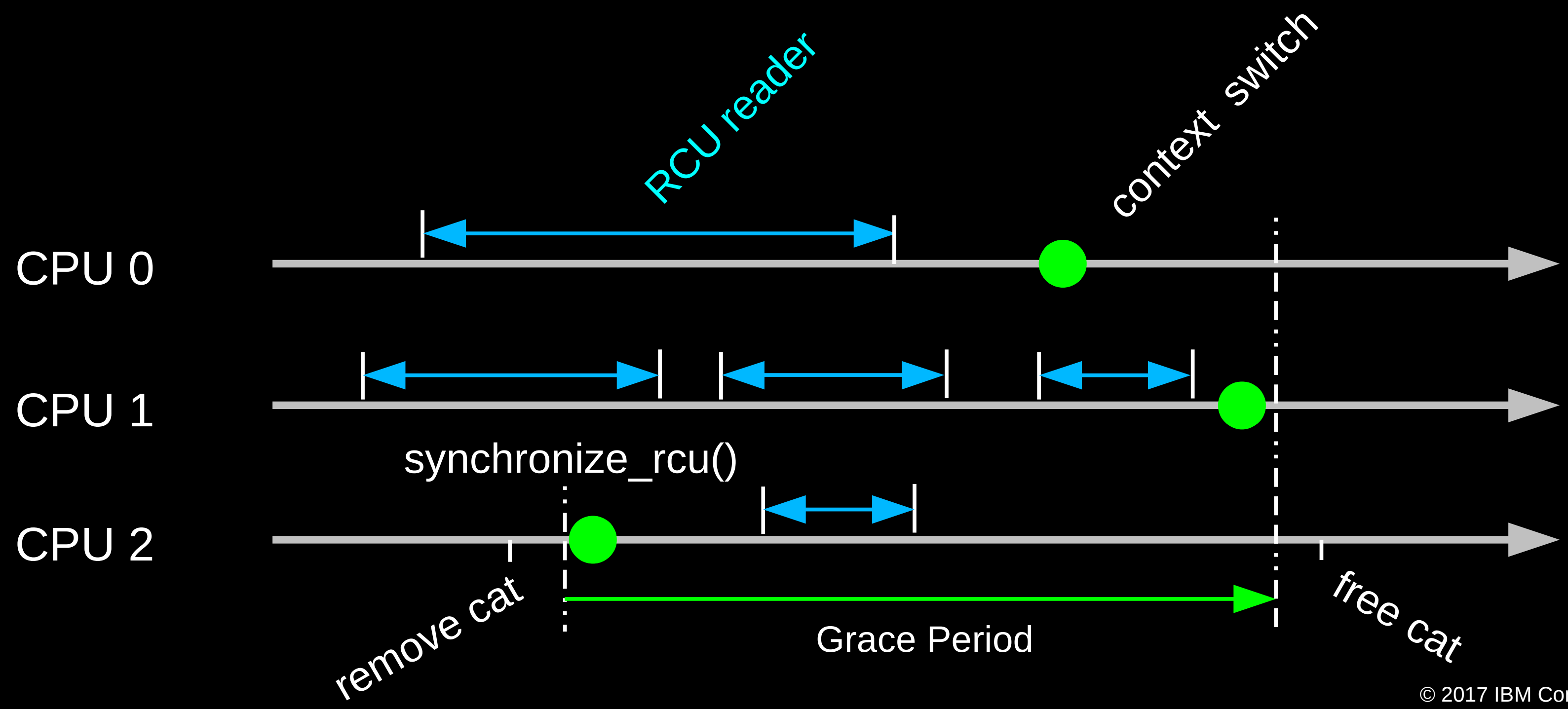
- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



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

RCU 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() and rcu_read_unlock() do not need to change machine state
  - Instead, they act on the developer, who must avoid blocking within RCU read-side critical sections

Synchronization Without Changing Machine State???

- But rcu_read_lock() and rcu_read_unlock() do not need to change machine state
  - Instead, they act on the developer, who must avoid blocking within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***

Synchronization Without Changing Machine State???

- But rcu_read_lock() and rcu_read_unlock() do not need to change machine state
  - Instead, they act on the developer, who must avoid blocking within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***
- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Access shared variables only while holding the corresponding lock”
  - “Access shared variables only within transactions”

Synchronization Without Changing Machine State???

- But rcu_read_lock() and rcu_read_unlock() do not need to change machine state
  - Instead, they act on the developer, who must avoid blocking within RCU read-side critical sections
- RCU is therefore ***synchronization via social engineering***
- As are all other synchronization mechanisms:
  - "Avoid data races"
  - "Access shared variables only while holding 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

RCU Is Specialized, And Does Not Stand Alone...

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

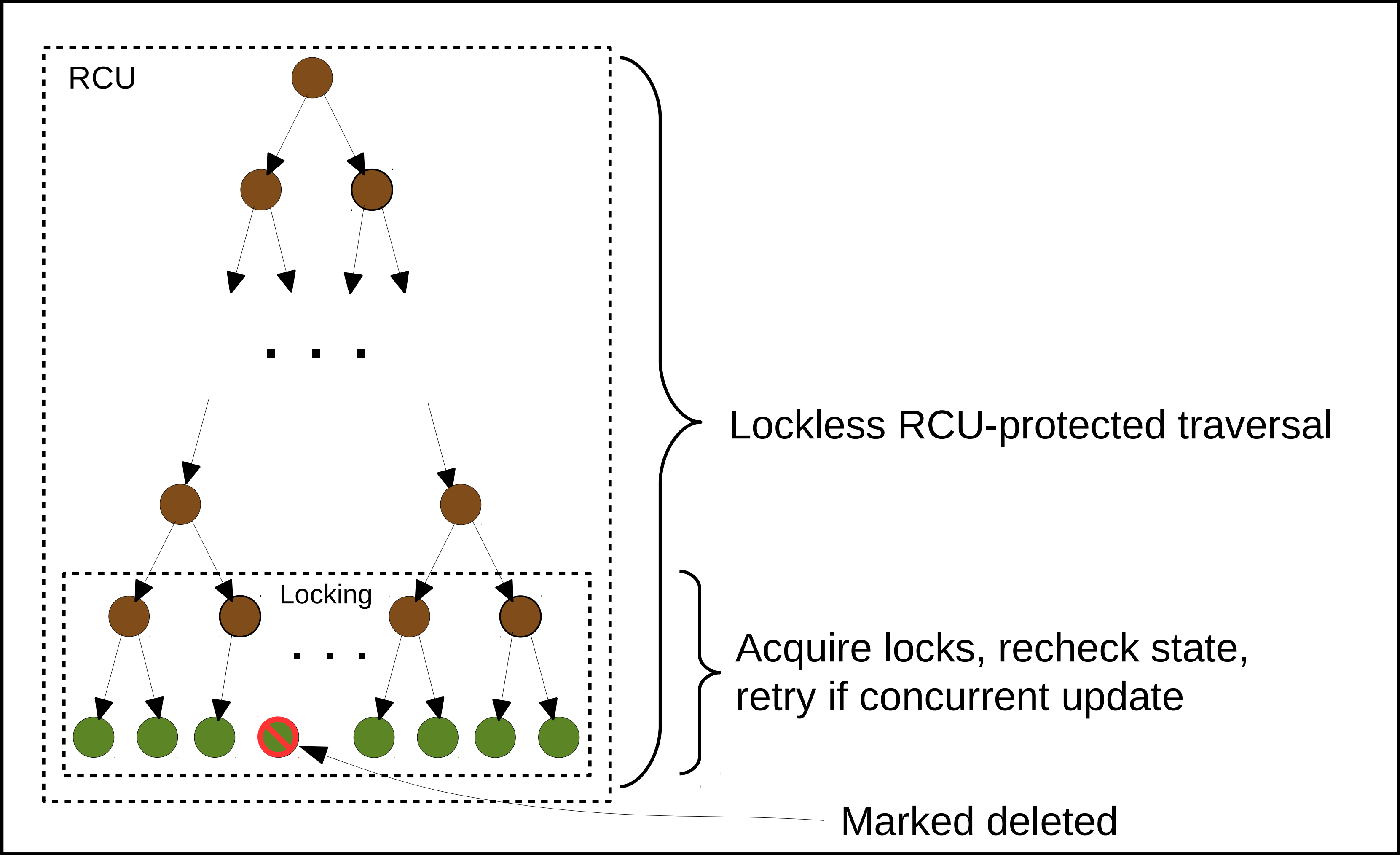
Read-Mostly, Need Consistent Data
(RCU Works OK)

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

Update-Mostly, Need Consistent Data
(RCU is ***Really*** Unlikely to be the Right Tool For The Job, But It Can:
(1) Provide Existence Guarantees For Update-Friendly Mechanisms
(2) Provide Wait-Free Read-Side Primitives for Real-Time Use)

Read-Only Traversal To Update Location

Deletion-Flagged Read-Only Traversal

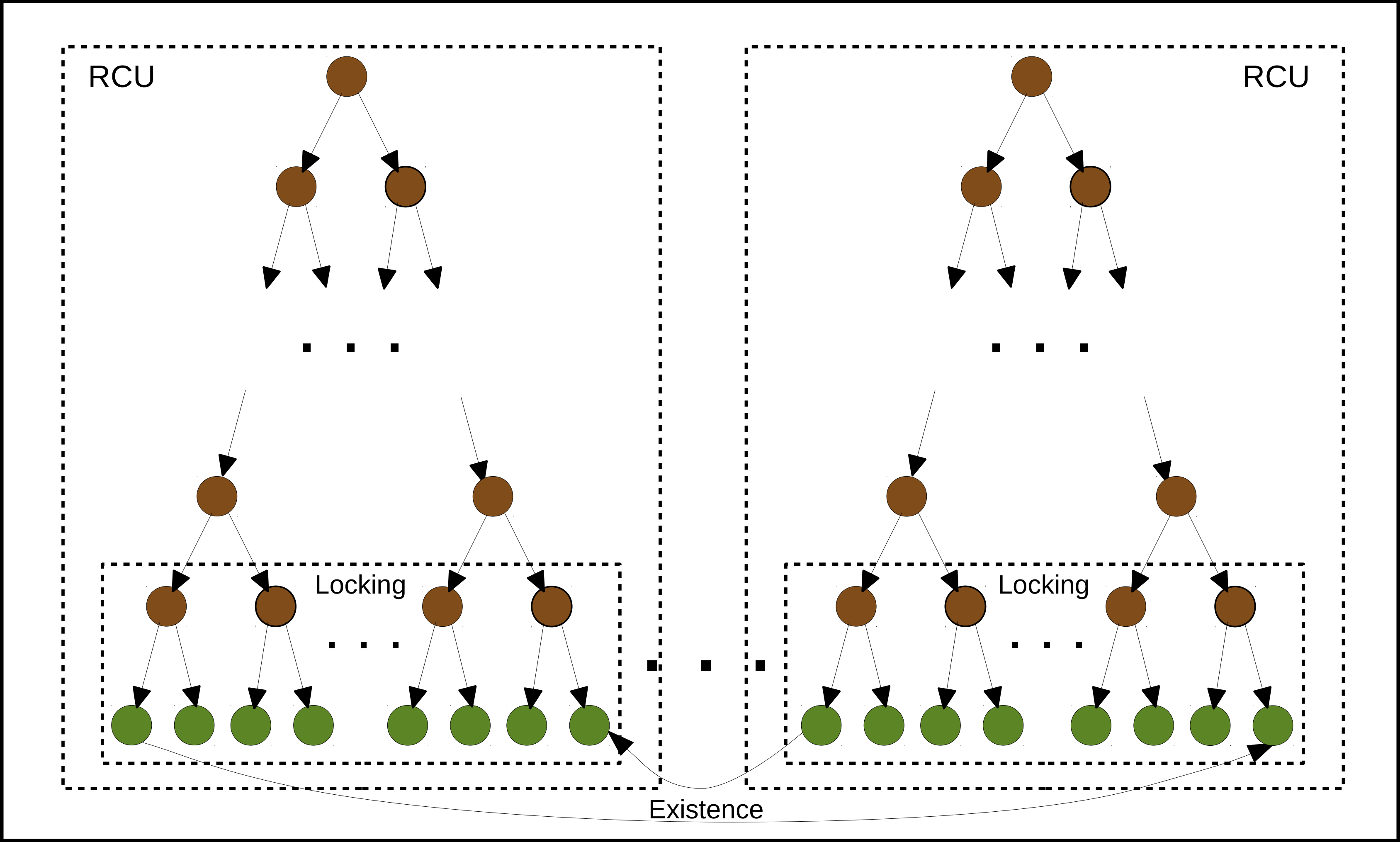


Read-Only Traversal To Location Being Updated

- 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:
  - David et al., “Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures”, Apr 2015 SIGPLAN Notices
  - 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

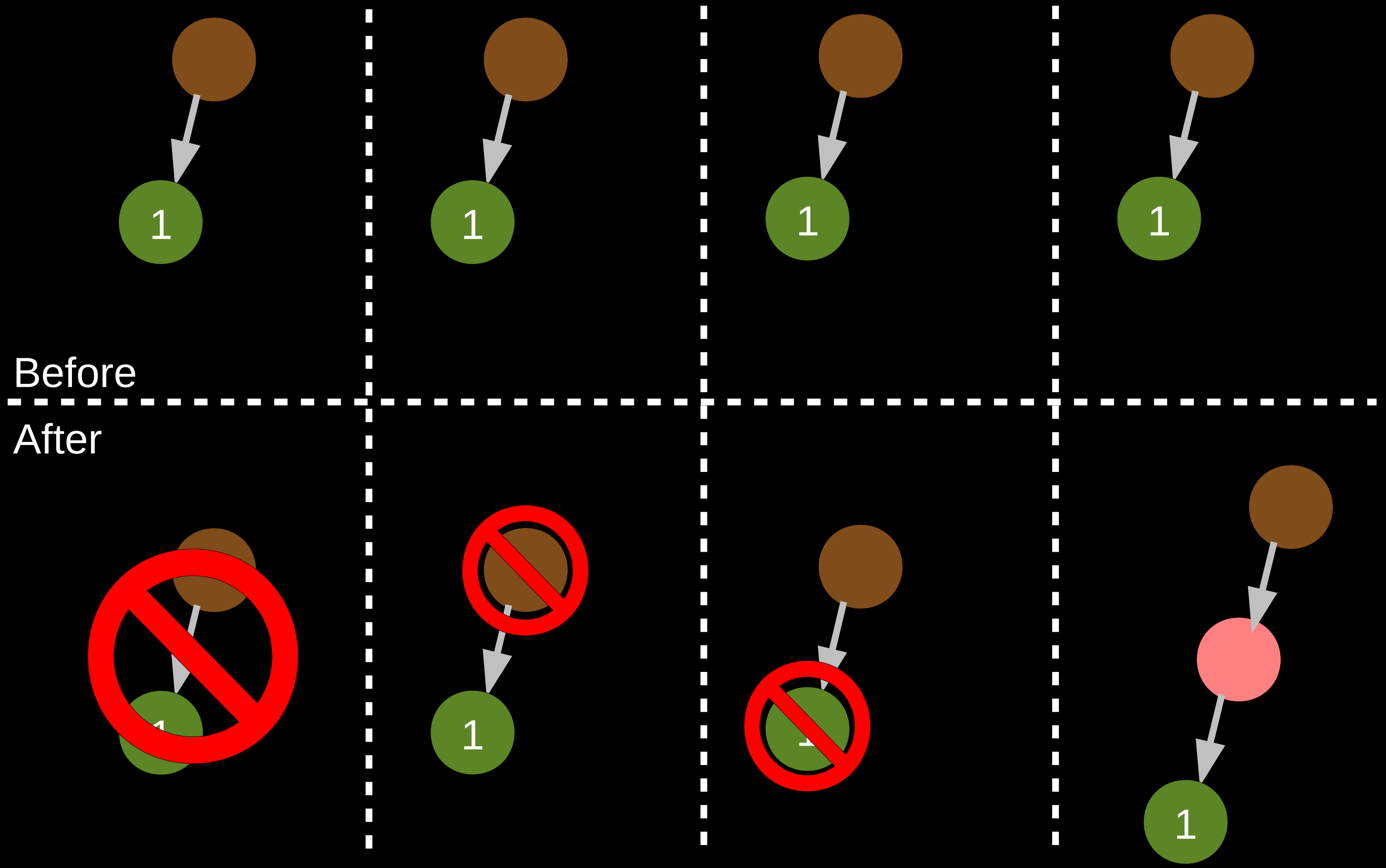
Issaquah Challenge: One Solution

Synchronization Regions for Binary Search Tree



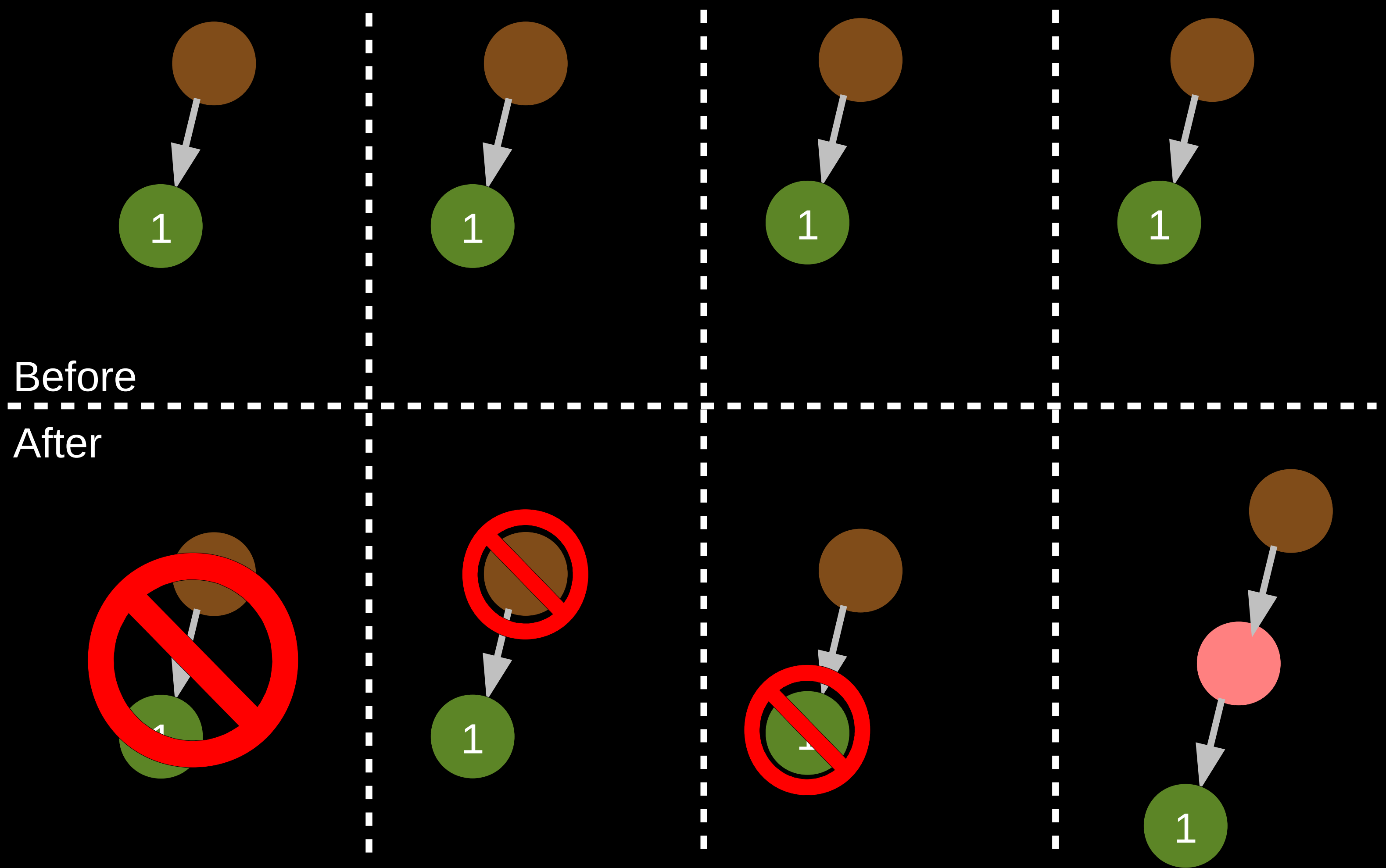
In many cases, can implement existence as simple wrapper!

Possible Upsets While Acquiring Locks...



What to do?
Drop locks and retry!!!

Possible Upsets While Acquiring Locks... But Independent of Atomic Moves!



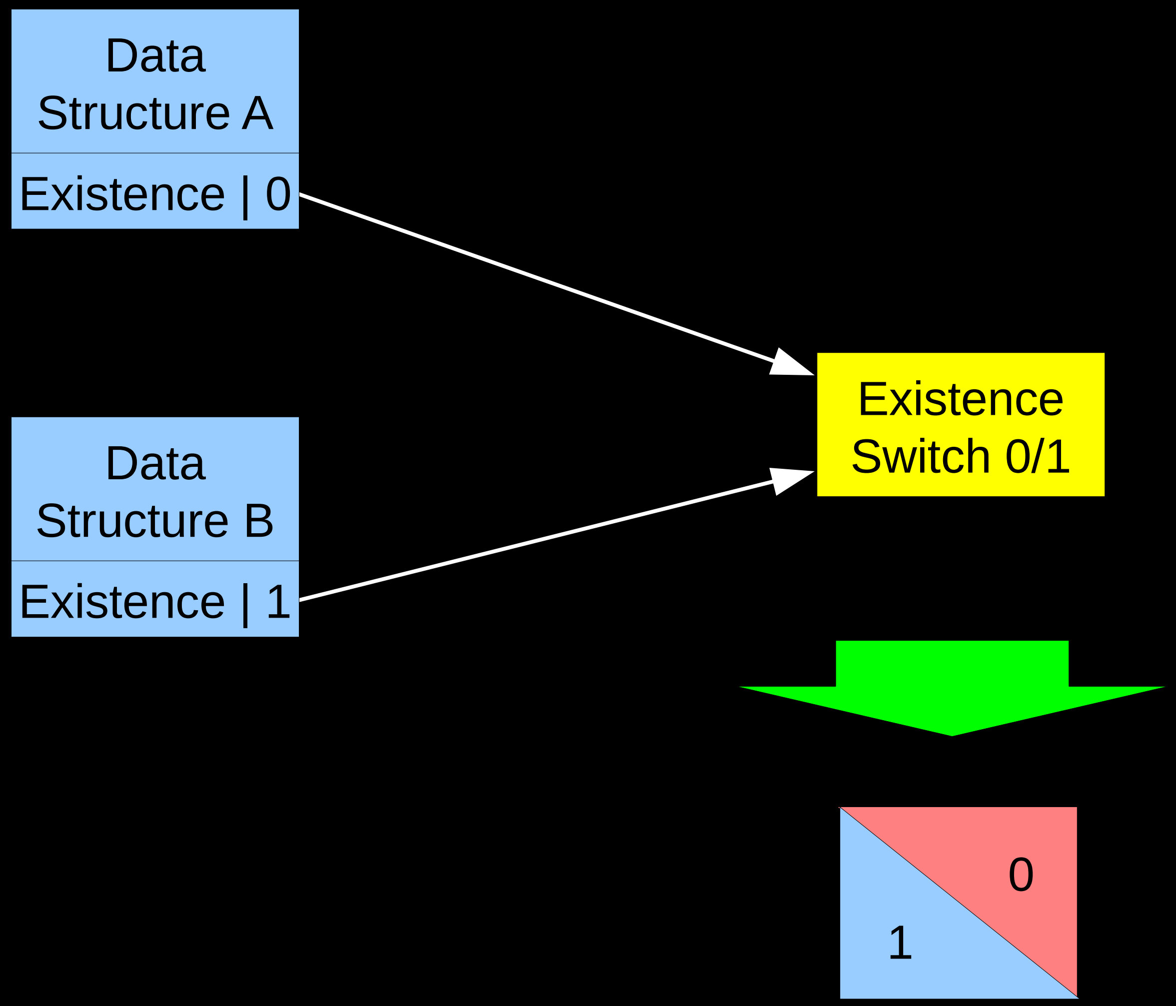
What to do?
Drop locks and retry!!!

Existence Structures

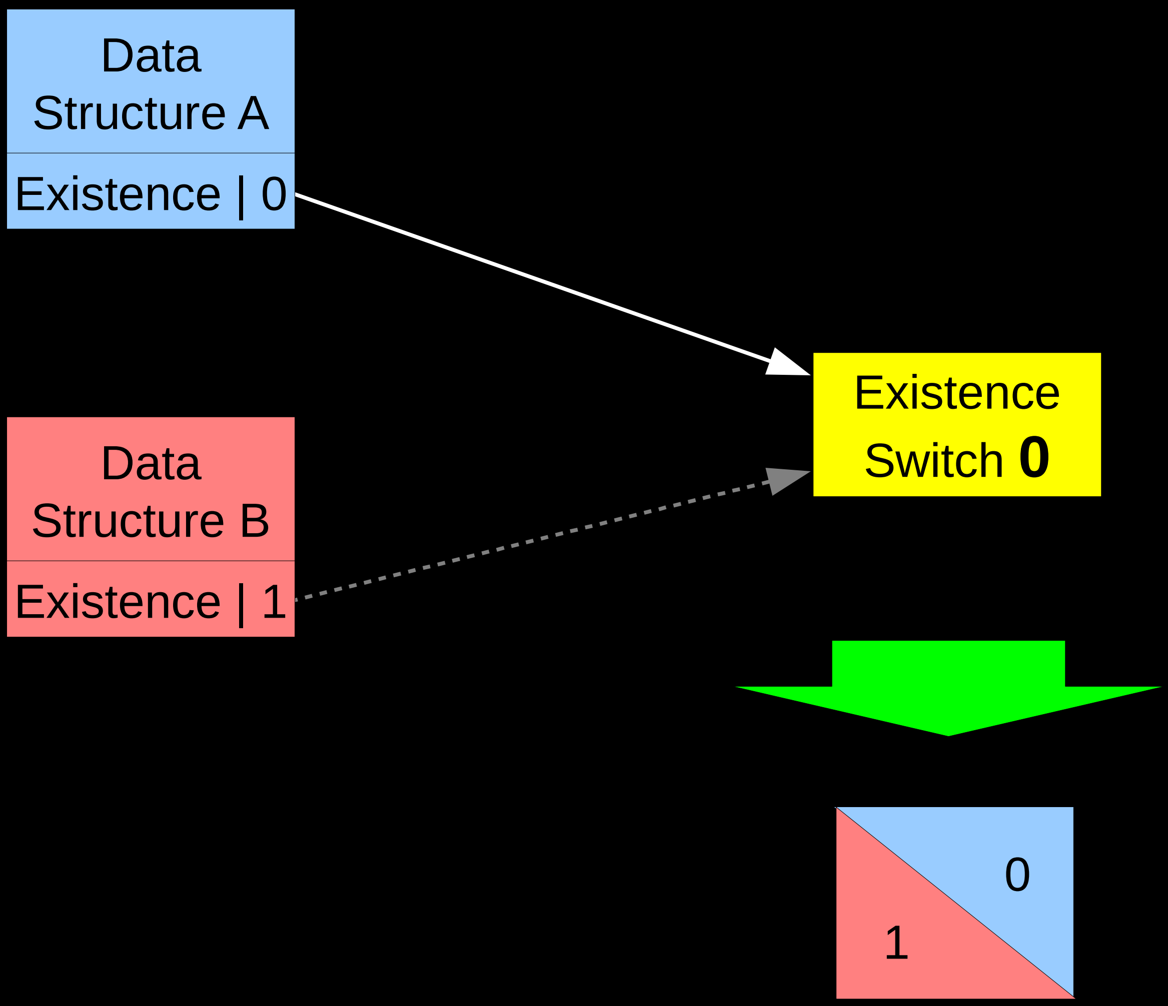
Existence Structures

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

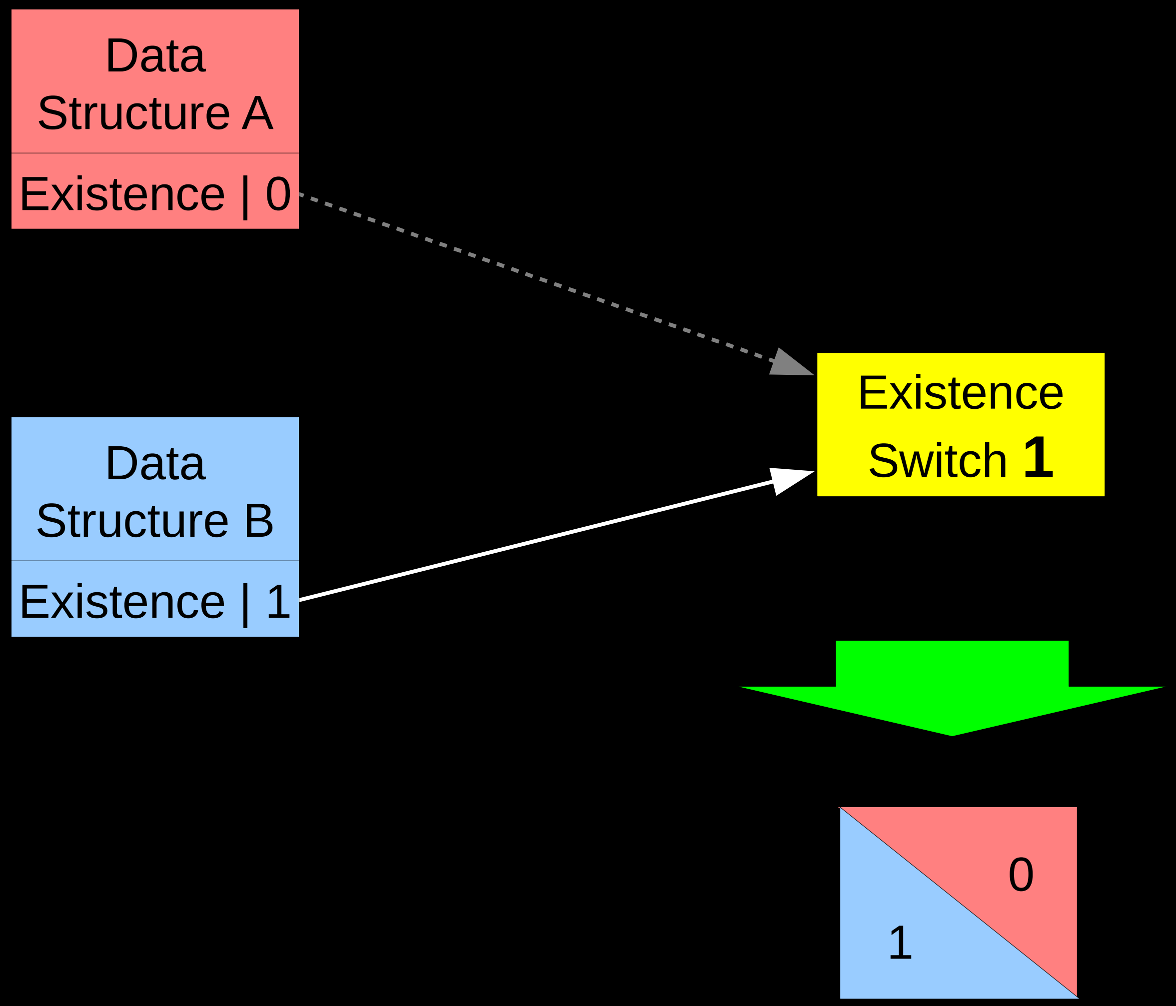
Example Existence Structure: Dmitry's Approach



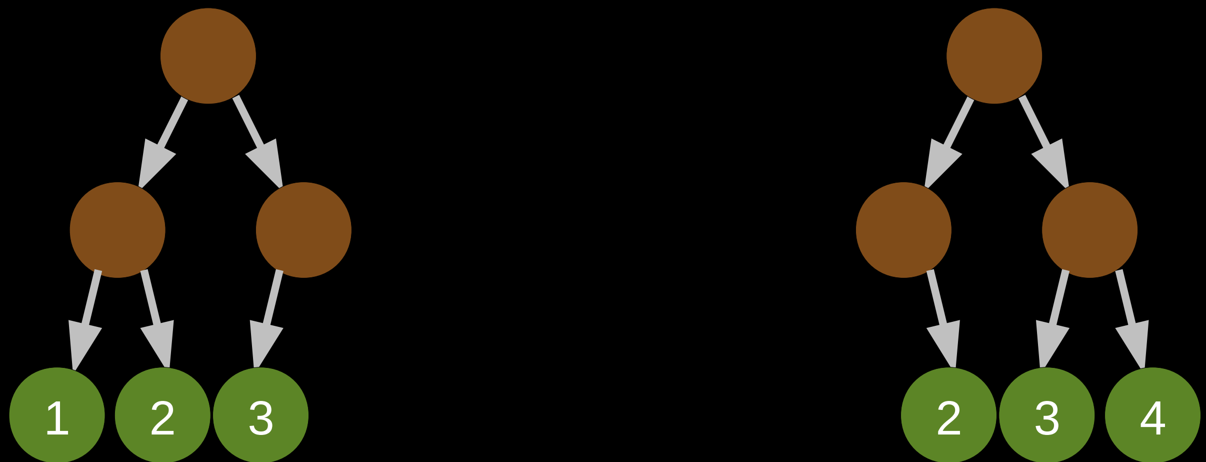
Example Existence Structure: Dmitry's Approach



Example Existence Structure: Dmitry's Approach

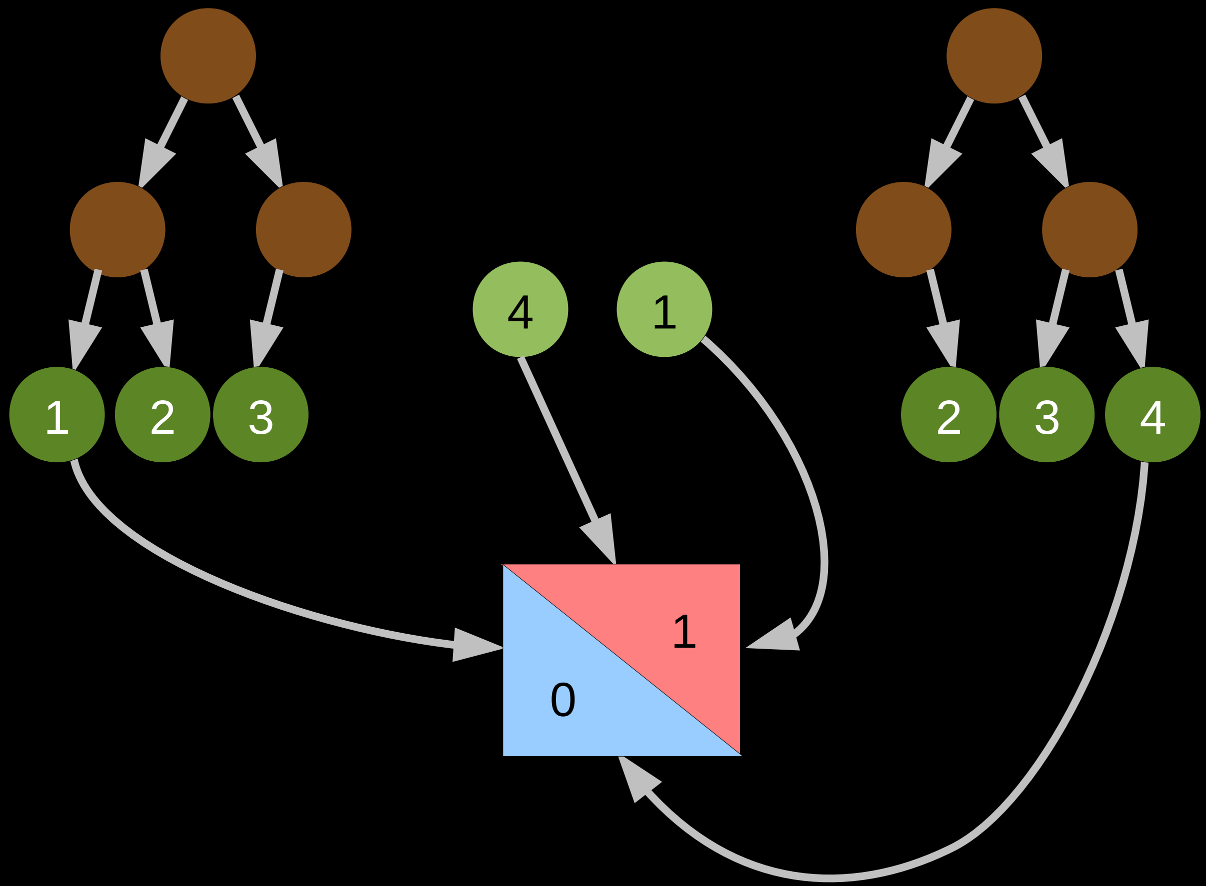


Abbreviated Existence Switch Operation (1/6)



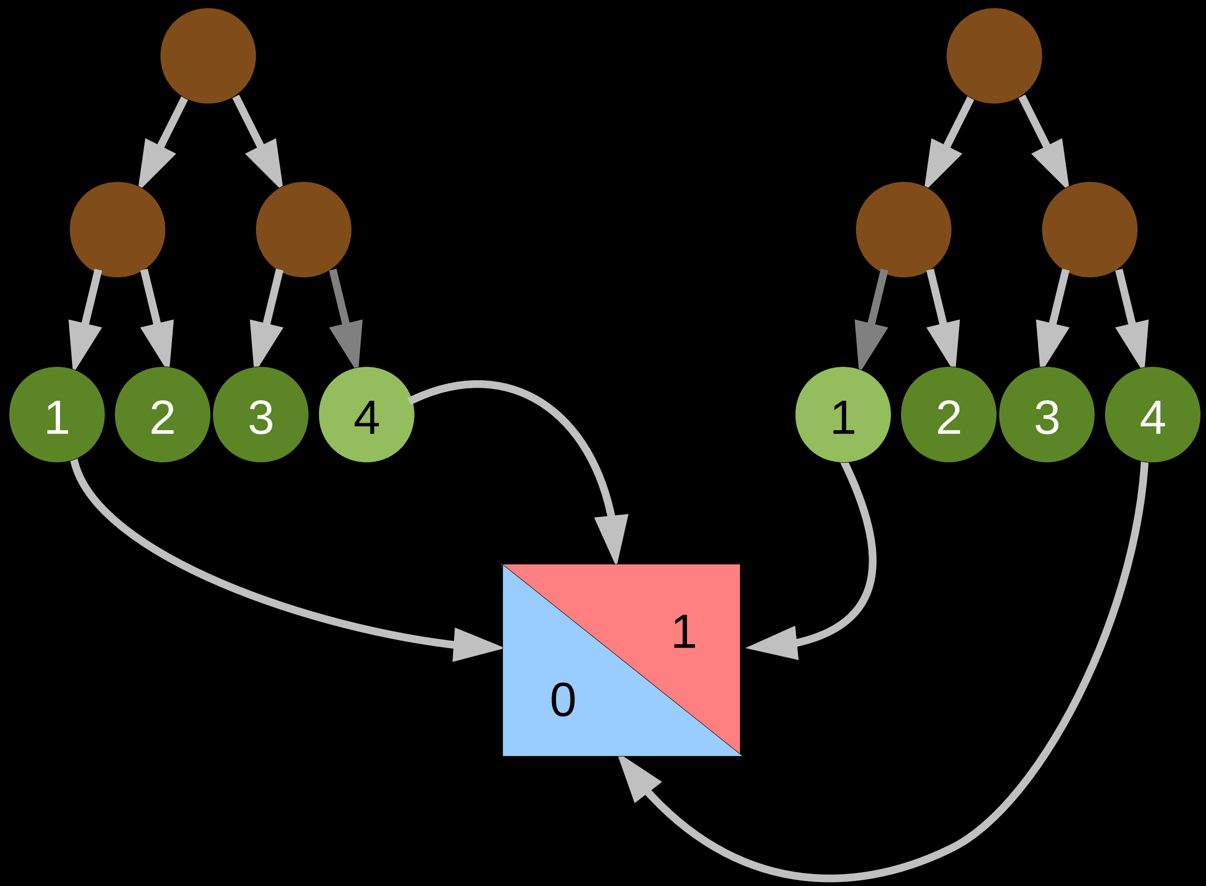
Initial state: First tree contains 1,2,3, second tree contains 2,3,4.
All existence pointers are NULL.

Abbreviated Existence Switch Operation (2/6)



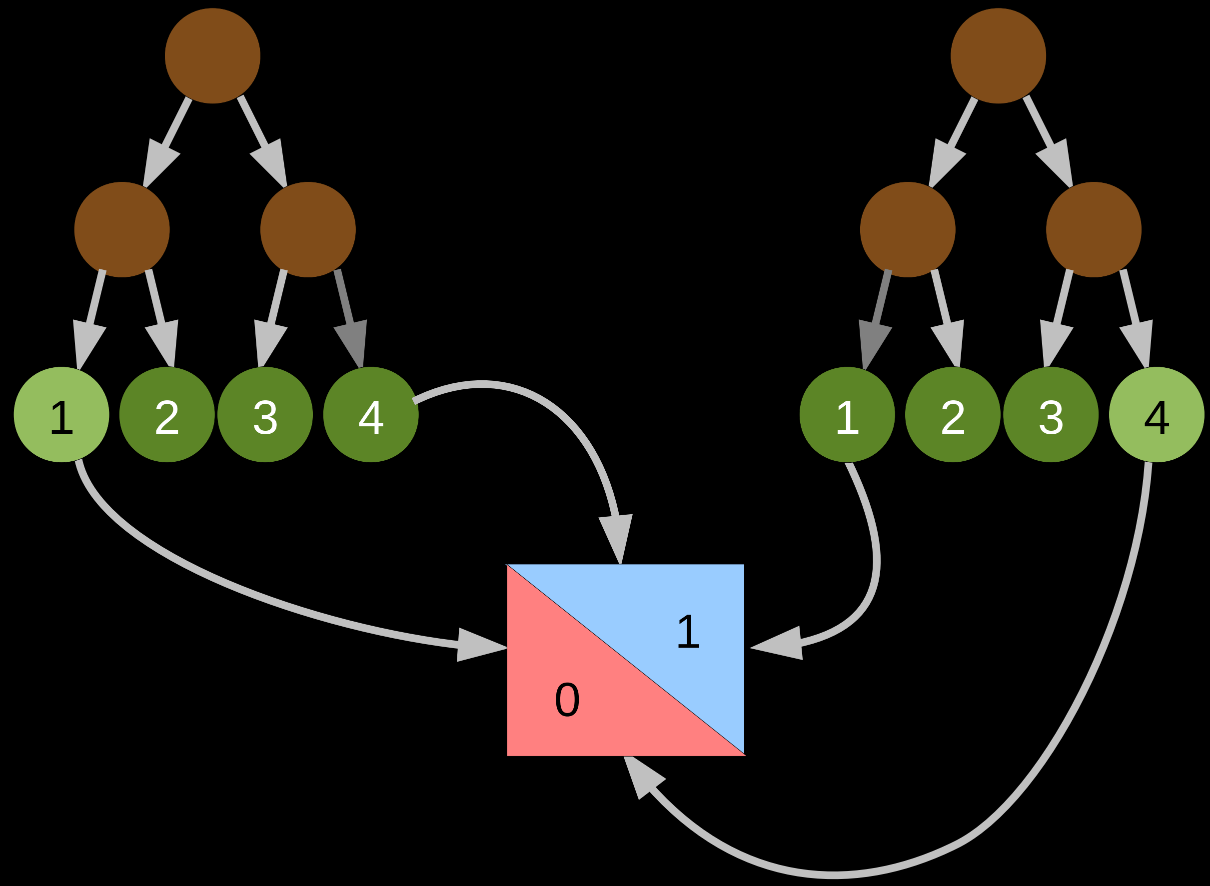
First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (3/6)



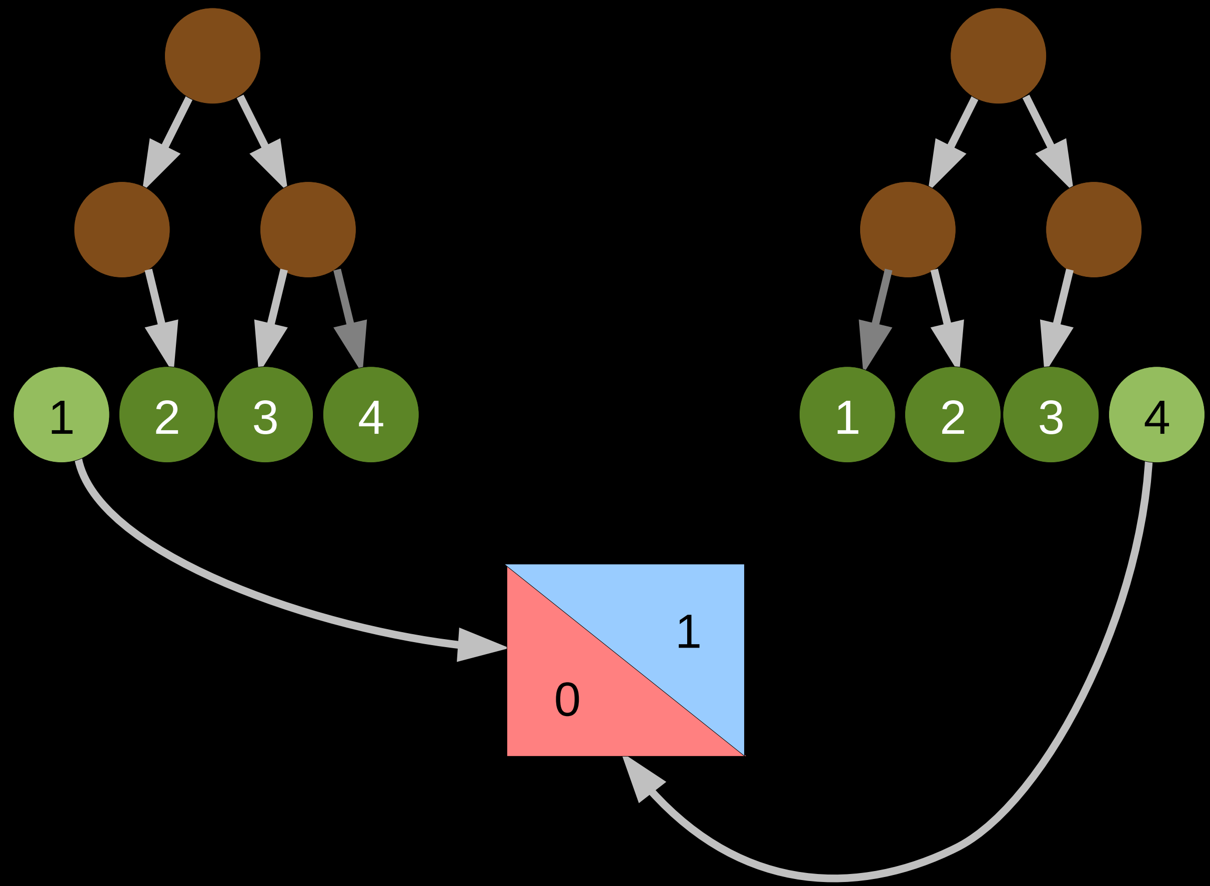
After insertion, same: First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (4/6)



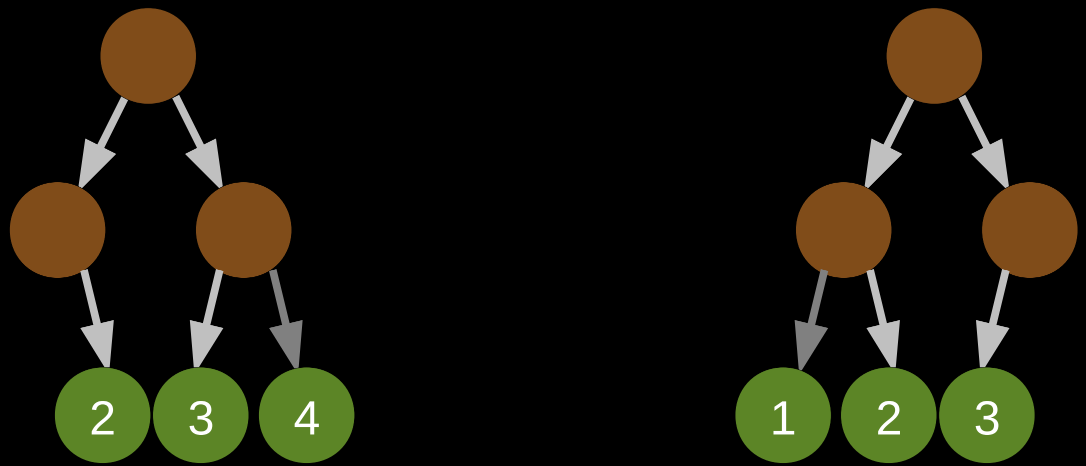
After existence switch: First tree contains 2,3,4, second tree contains 1,2,3.
Transition is single store, thus atomic! (But lookups need barriers in this case.)

Abbreviated Existence Switch Operation (5/6)



Unlink old nodes and existence structure
(Now automated!)

Abbreviated Allegiance 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
    - Pointer tag indicates outgoing (0) or incoming (1)
    - 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:

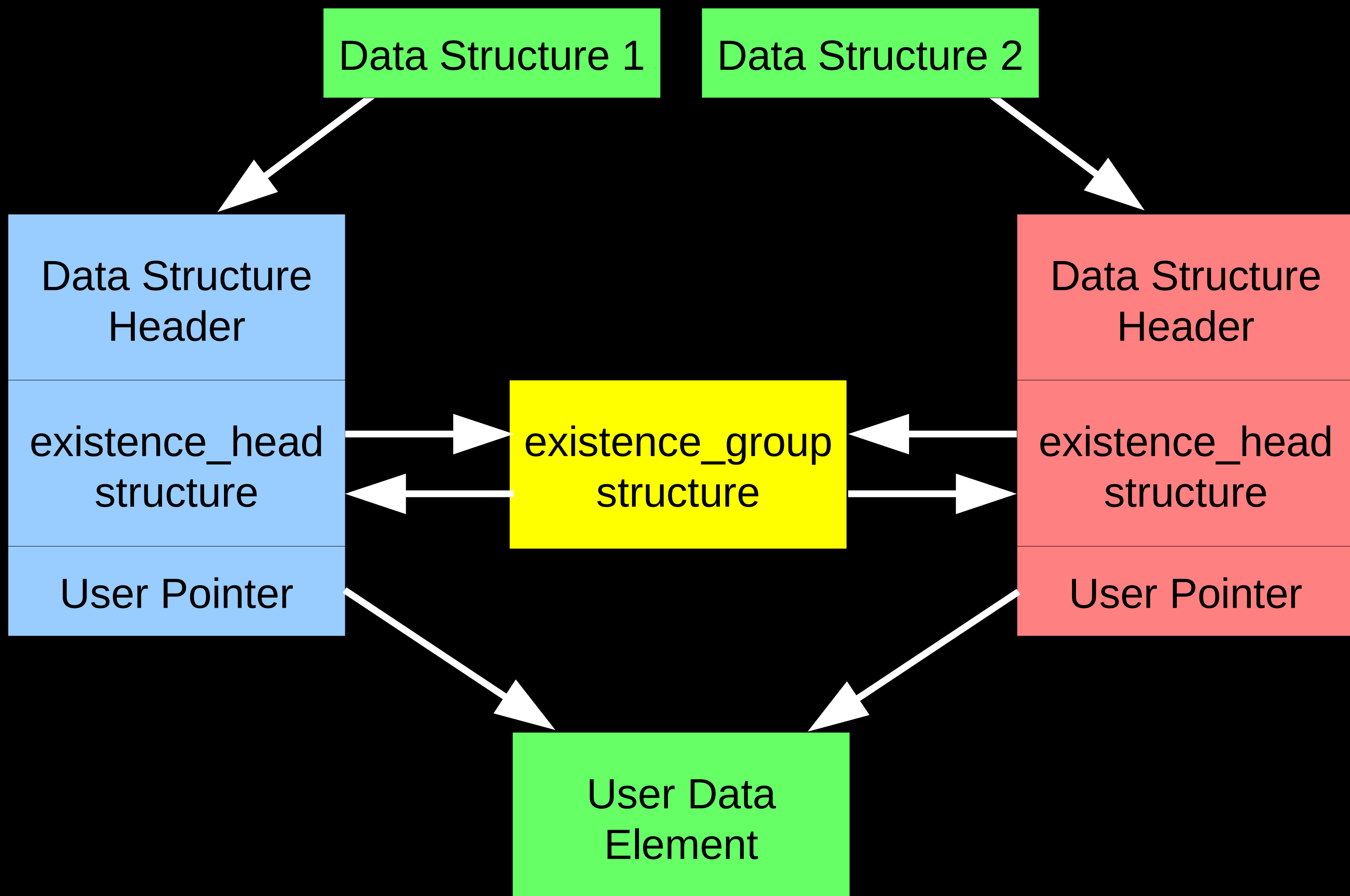
```
int existence_head_init_incoming(struct existence_head *ehp,
                                 struct existence_group *egp,
                                 int (*eh_add)(struct existence_head *ehp),
                                 void (*eh_remove)(struct existence_head *ehp),
                                 void (*eh_free)(struct existence_head *ehp))
```

- int existence_head_set_outgoing:

```
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

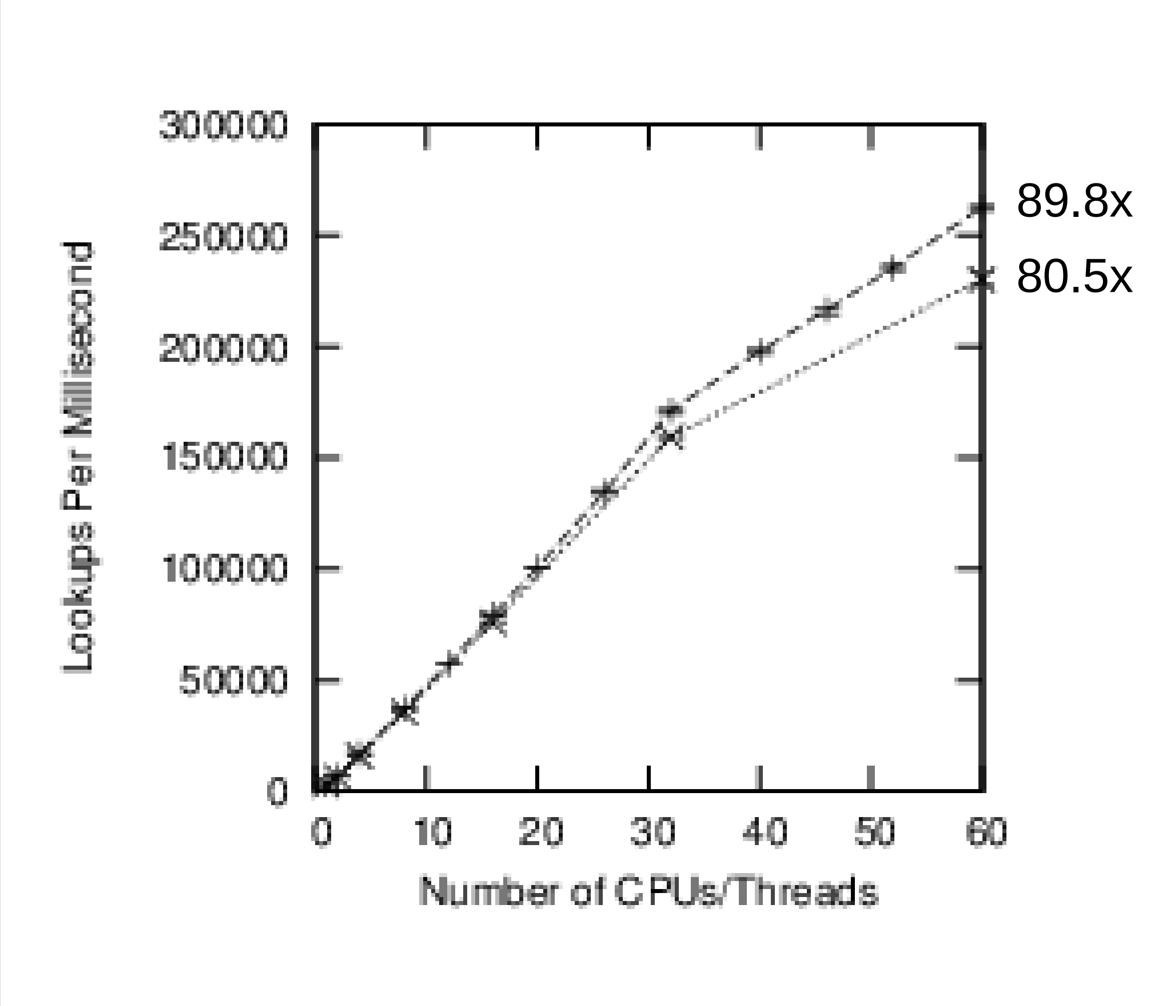


User data element atomically moving from data structure 1 to 2, which can be different types of data structures

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



LCA

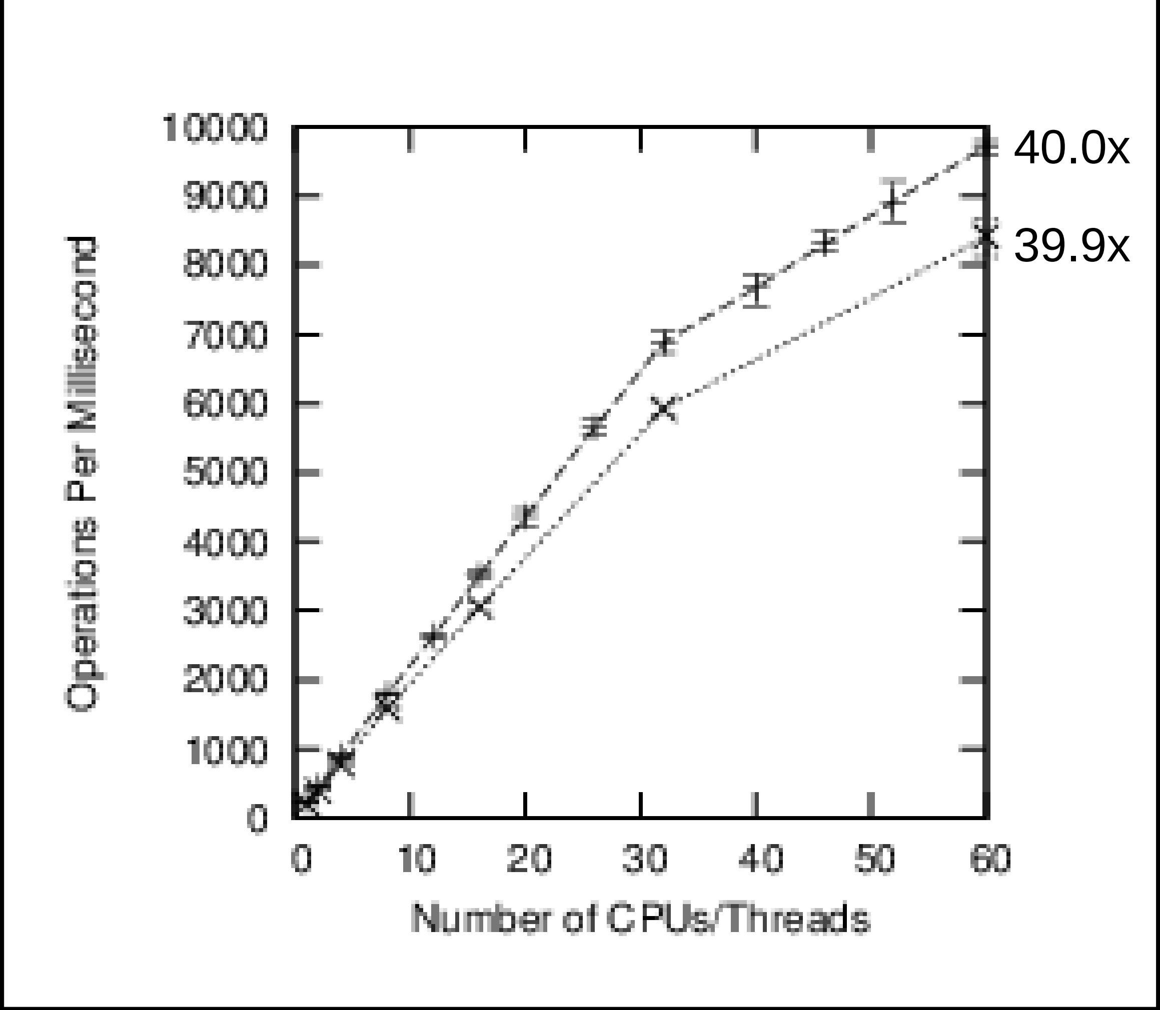
CPPCON

100% lookups

Super-linear as expected based on range partitioning

(Hash tables about 3x faster)

Existence Structures: Performance and Scalability

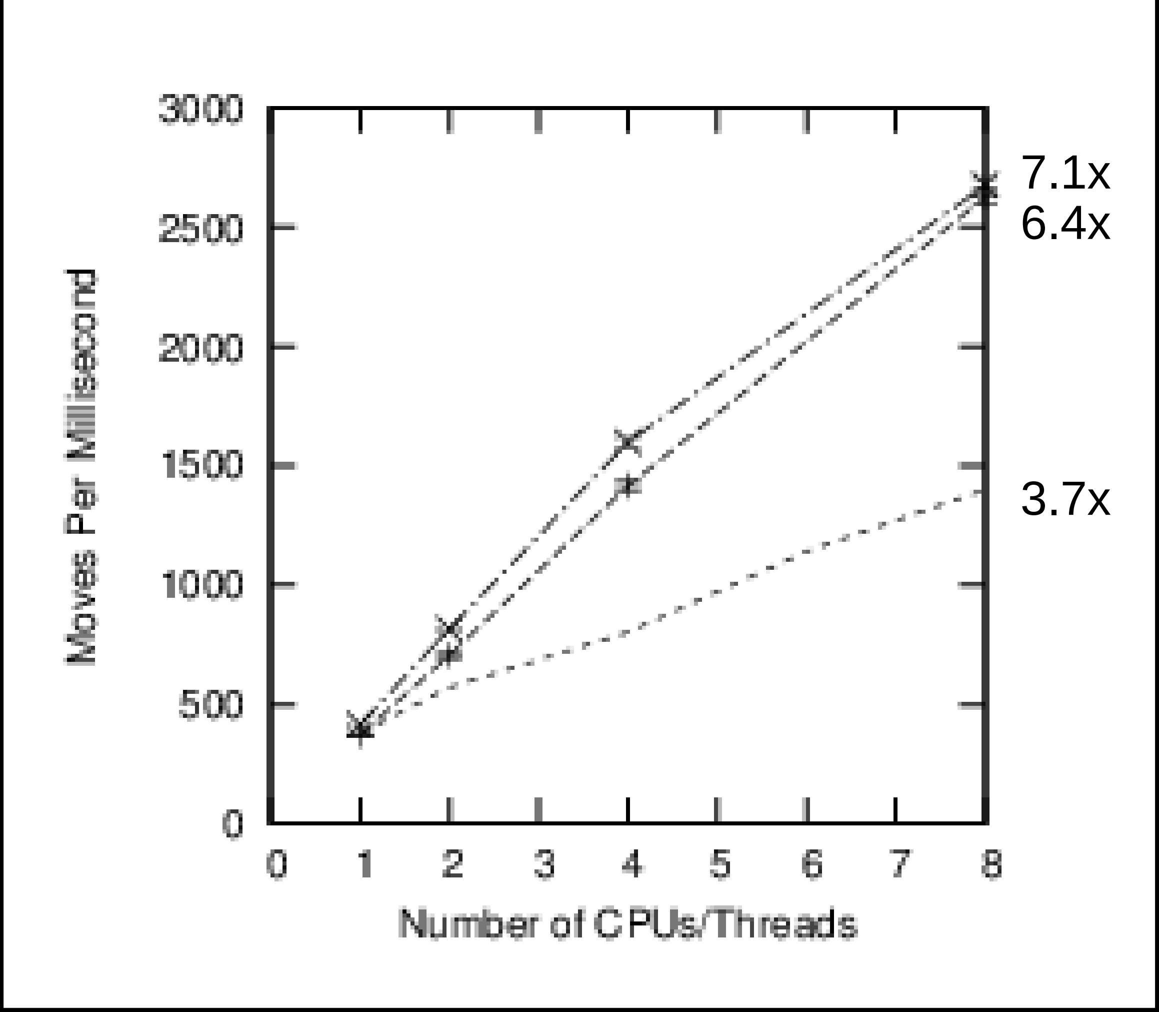


40.0x LCA

39.9x 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



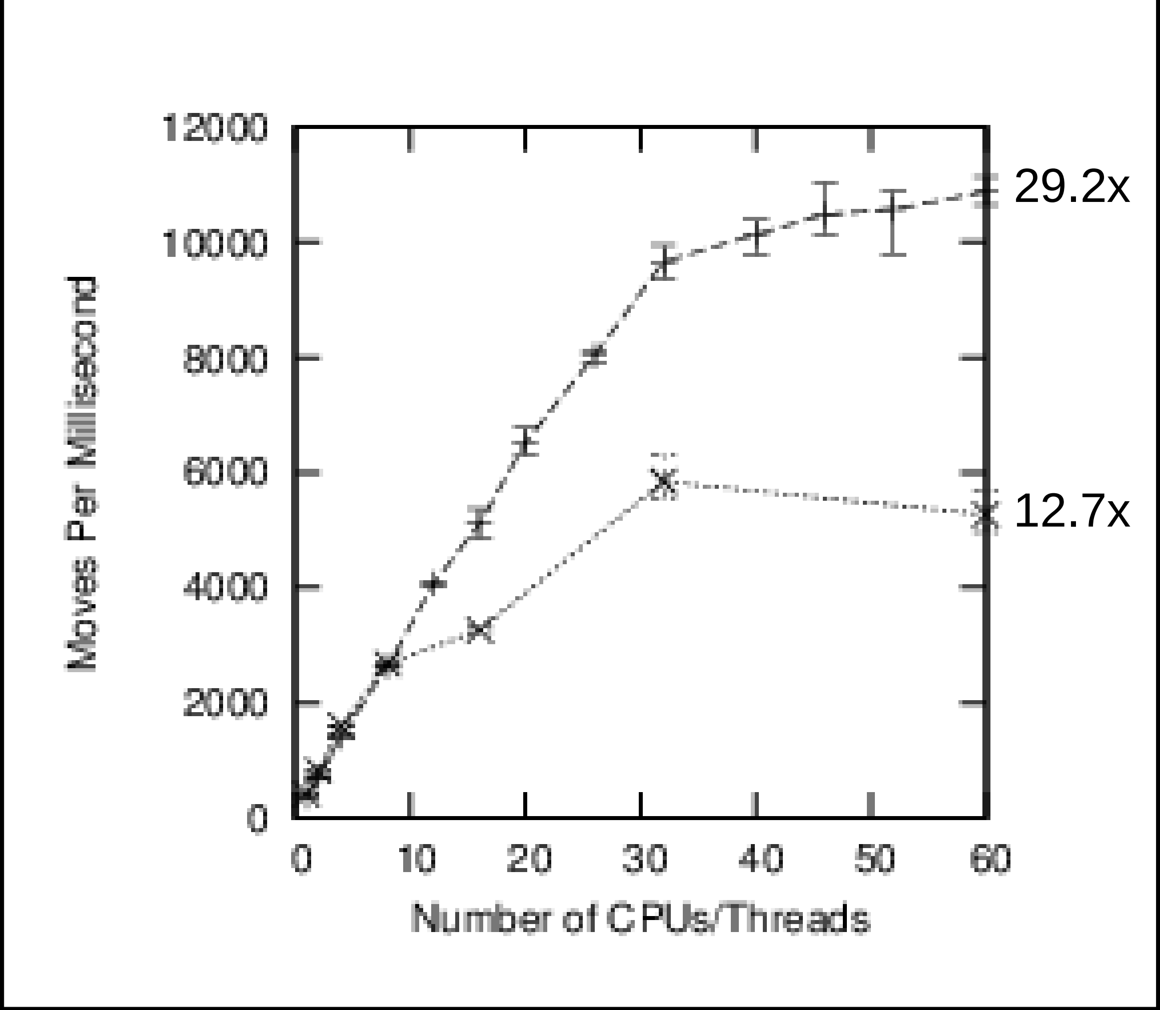
LCA

CPPCON

N4037

100% moves (worst case)

Existence Structures: Performance and Scalability



29.2x LCA

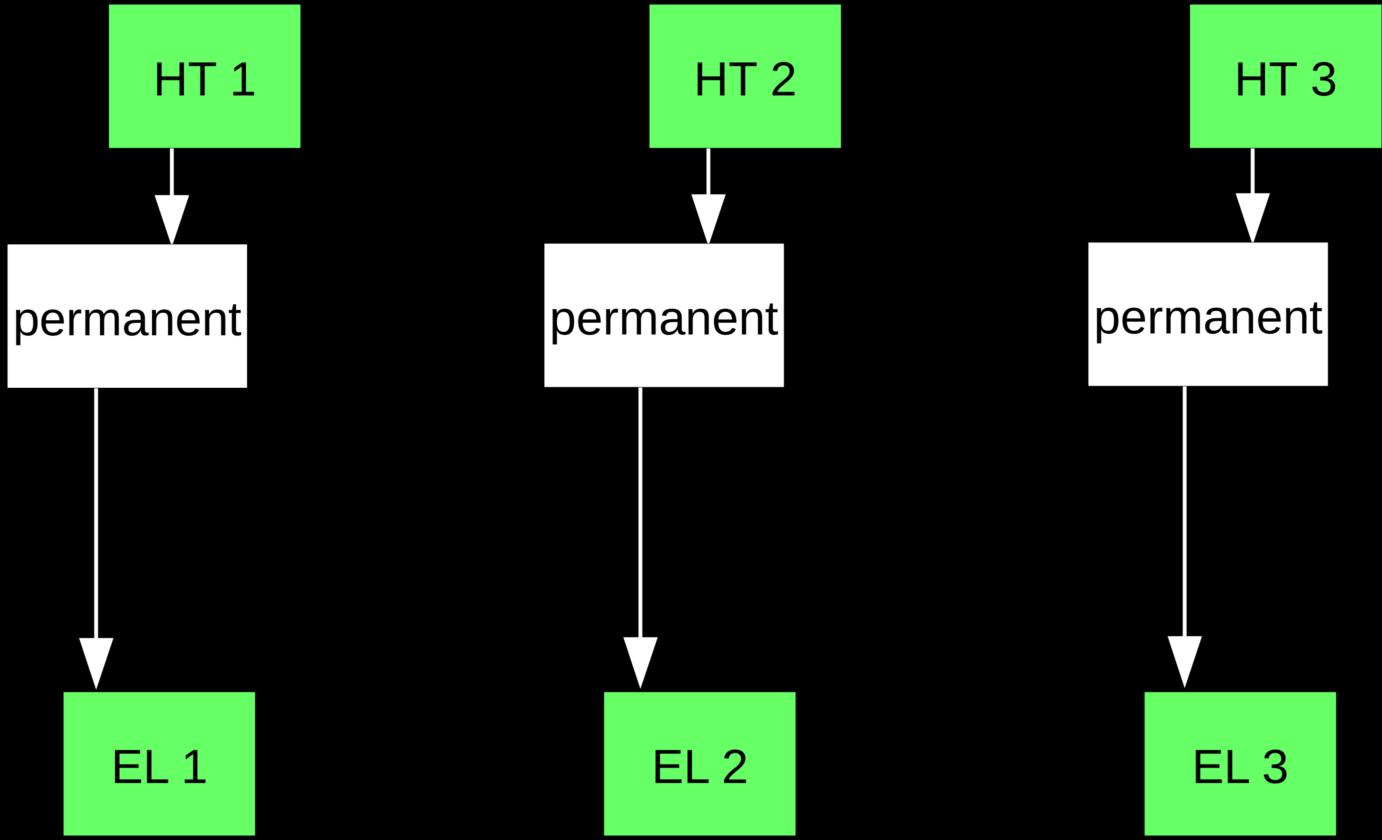
12.7x CPPCON

100% moves: Still room for improvement!
But at least we are getting positive scalability...

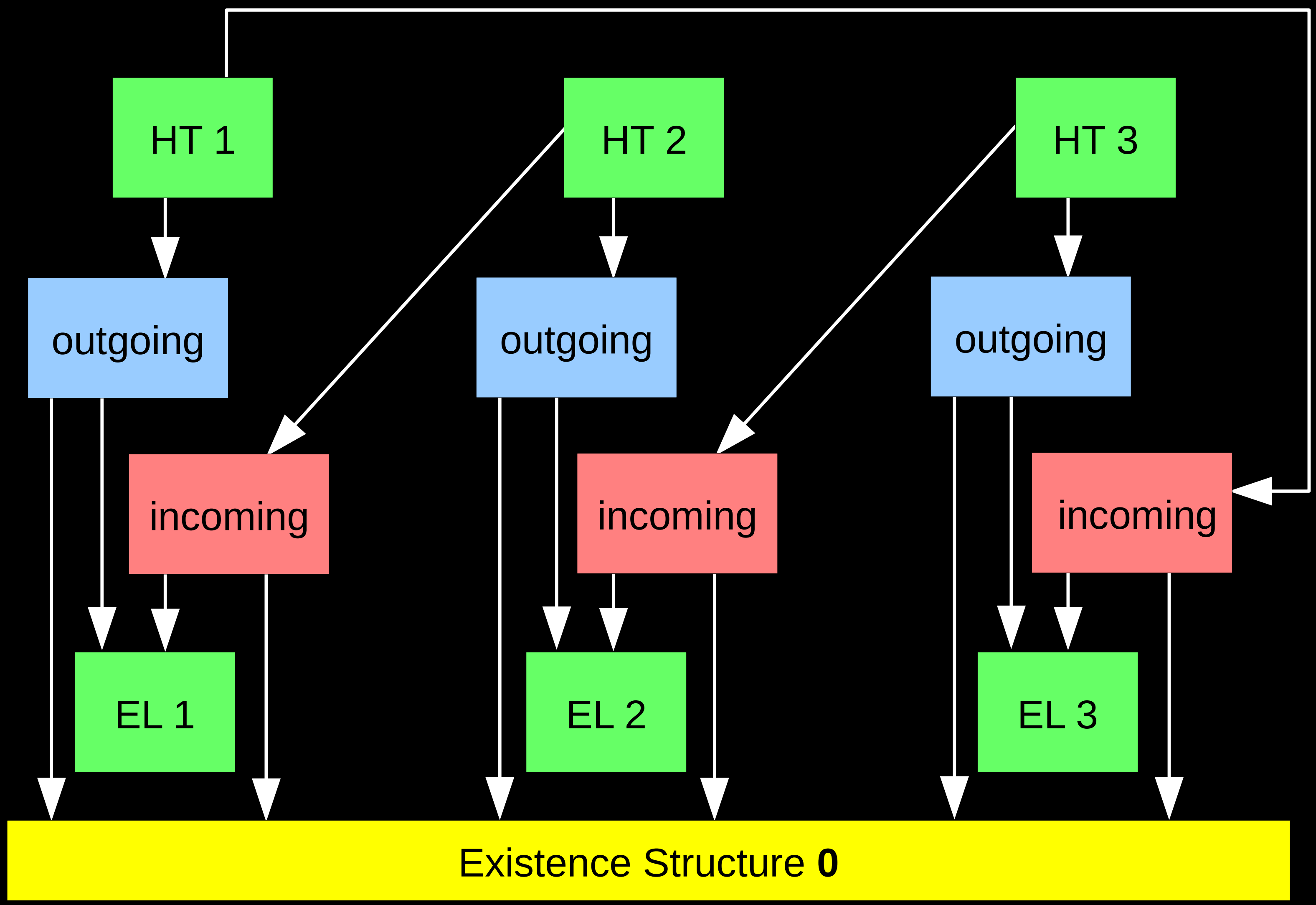
But Requires Modifications to Existing Algorithms

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

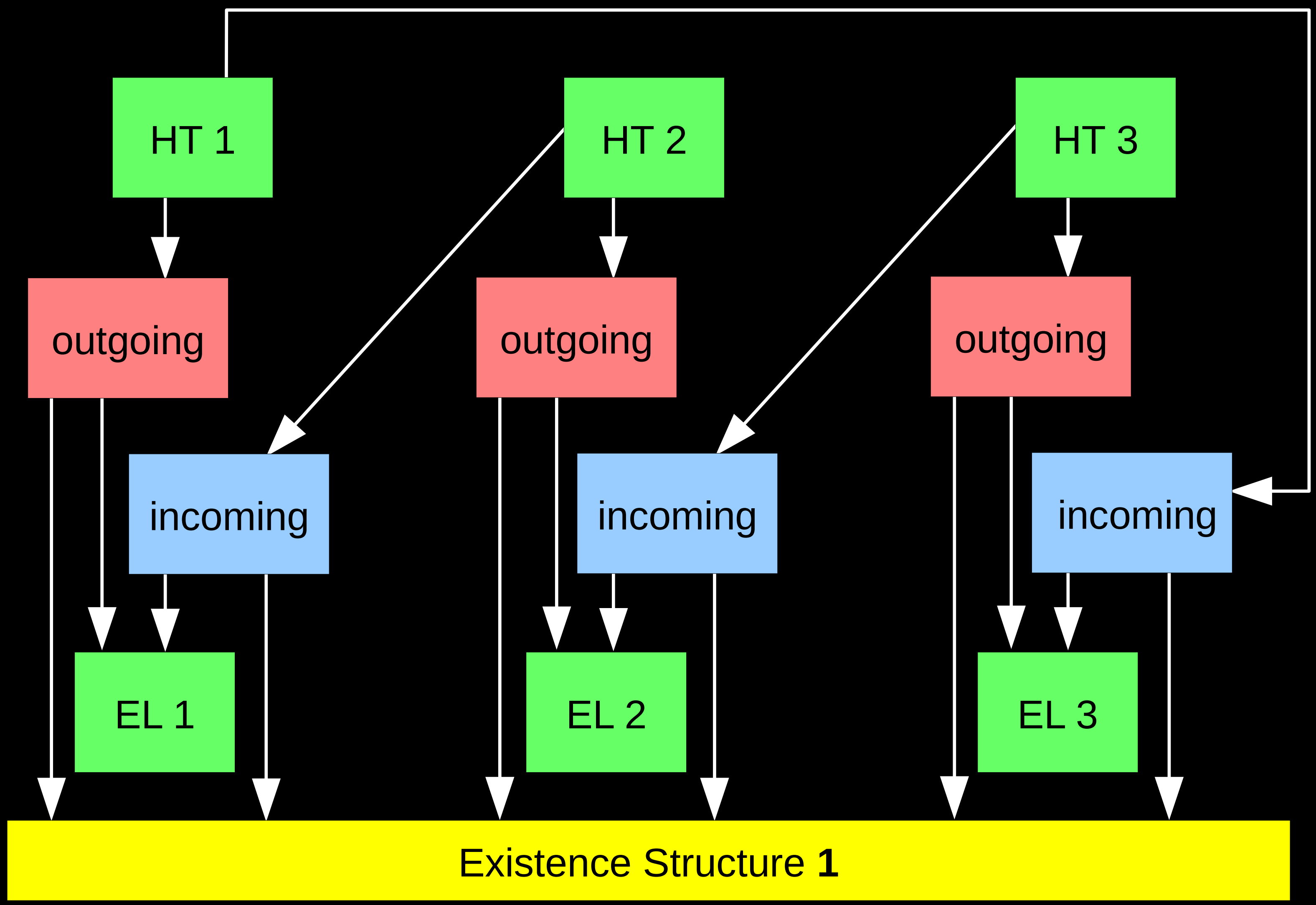
Rotate 3 Elements Through 3 Hash Tables (1/4)



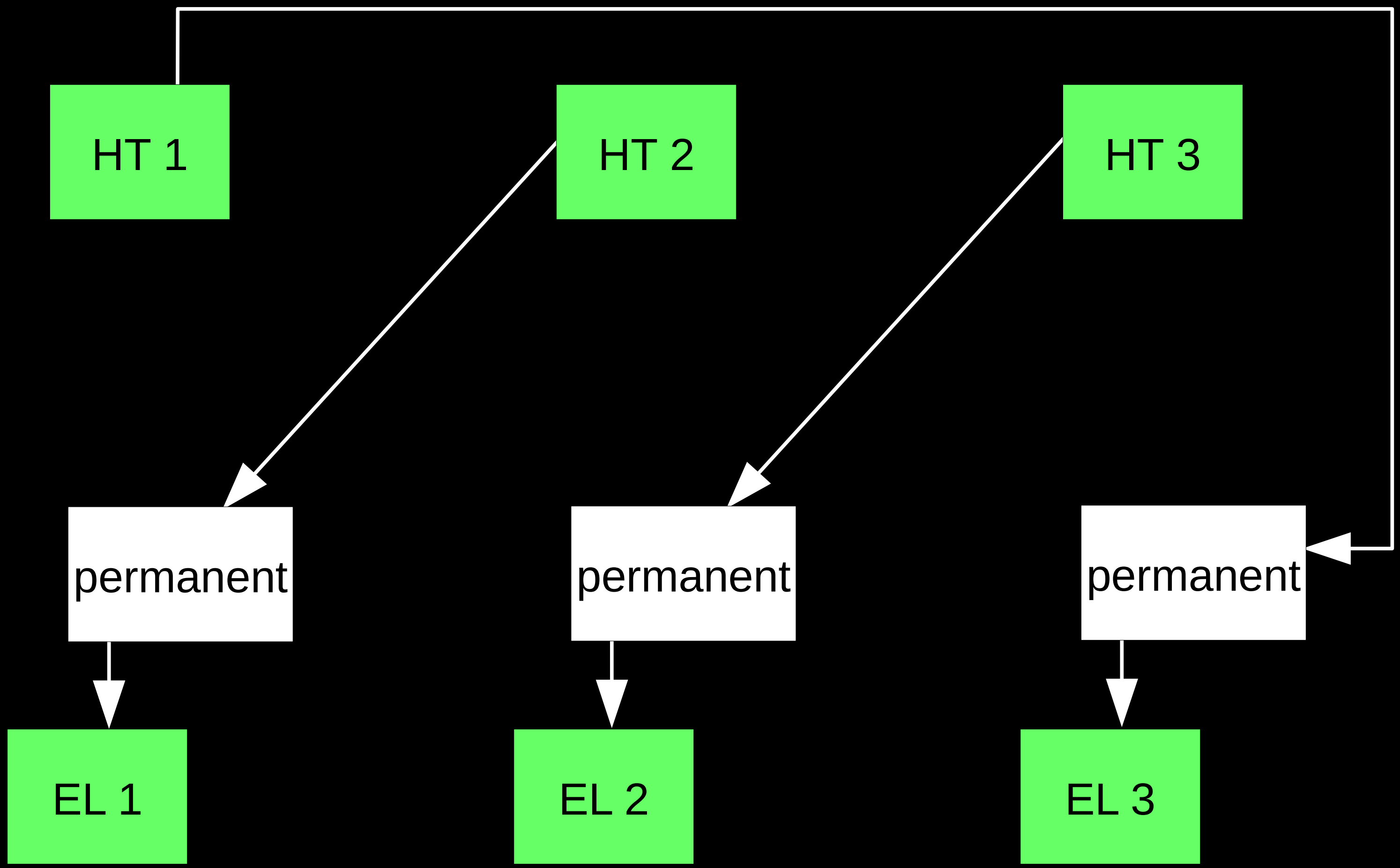
Rotate 3 Elements Through 3 Hash Tables (2/4)



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!

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

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

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

Update-Mostly, Need Consistent Data
(RCU is ***Really*** Unlikely to be the Right Tool For The Job, But It Can:
(1) Provide Existence Guarantees For Update-Friendly Mechanisms
(2) Provide Wait-Free Read-Side Primitives for Real-Time Use)

Opportunity to improve the infrastructure!

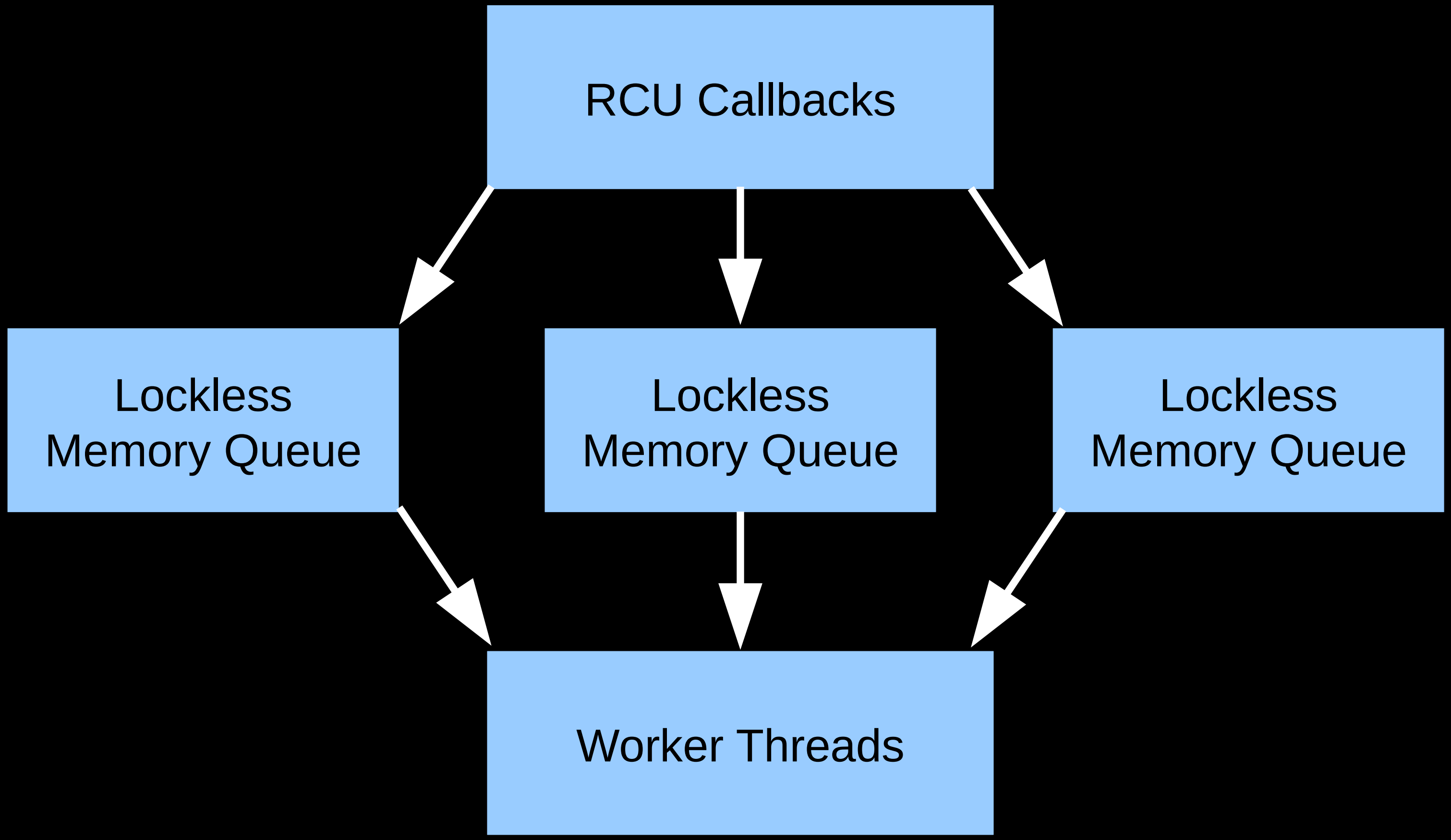
New Age Existence Structures: Towards Scalability

- "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!!!
    - 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

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!!!
    - 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

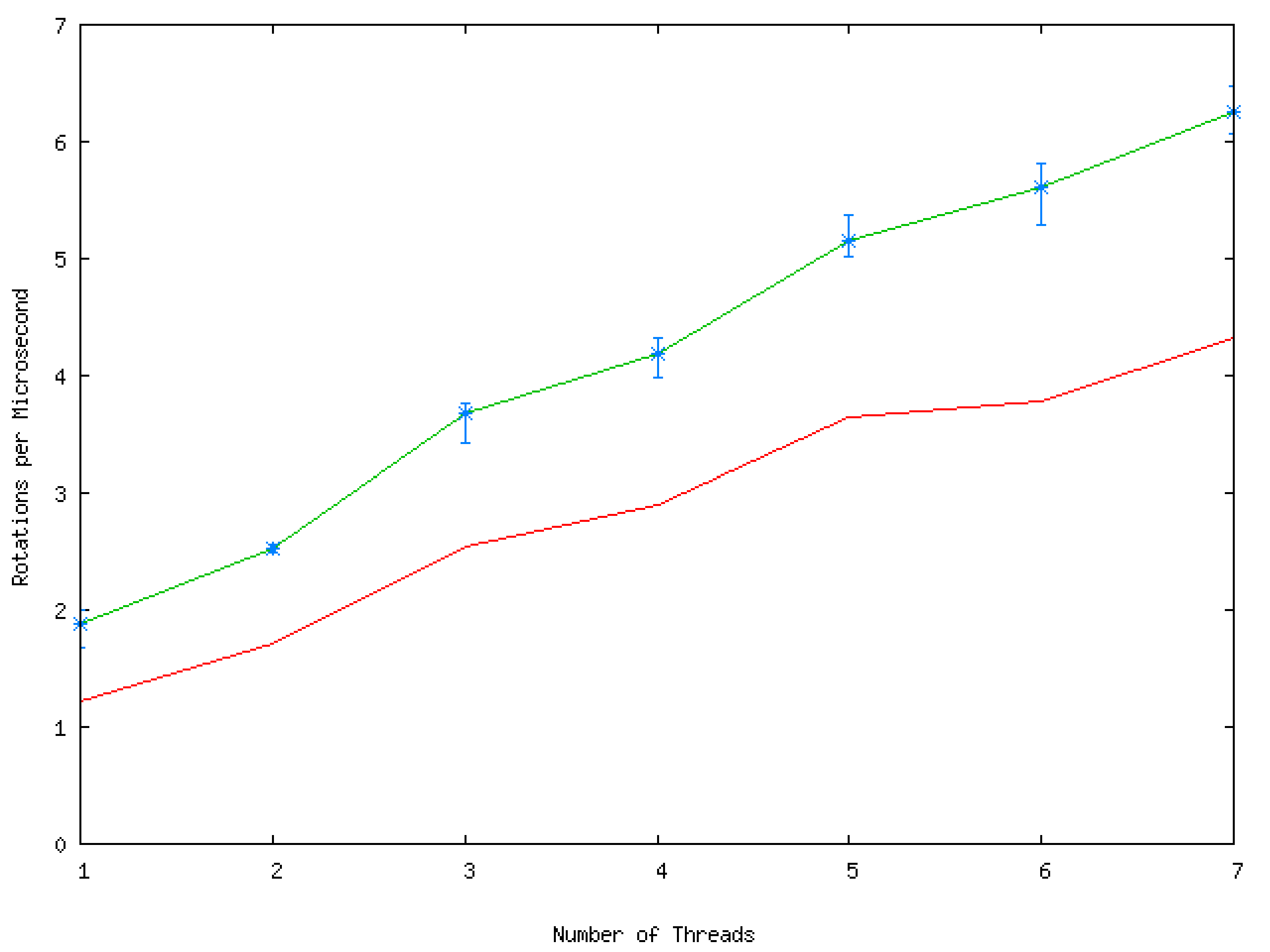
Specialized Producer/Consumer Allocator



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?



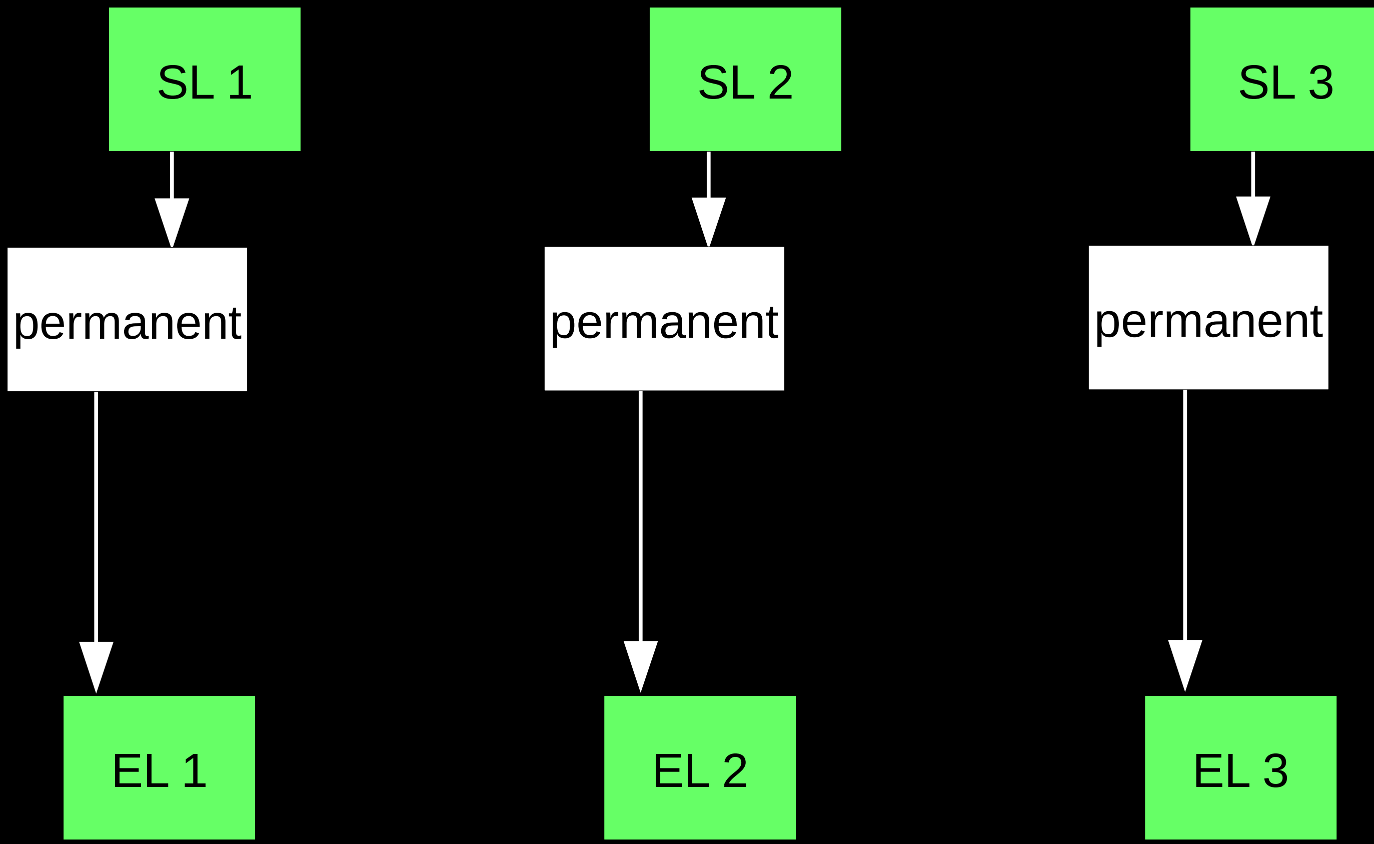
CPPCON

ACM App.

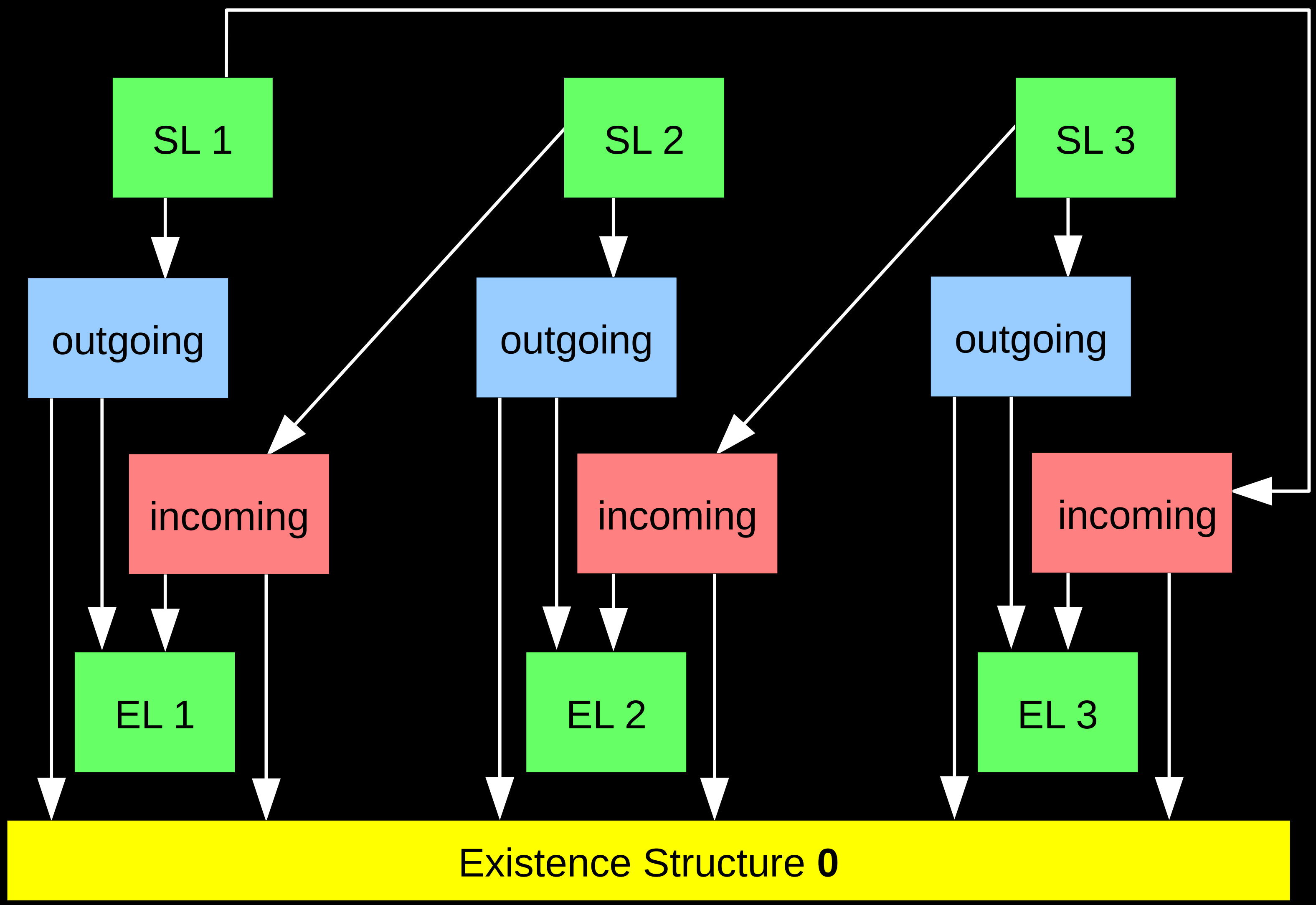
Some improvement, but still not spectacular
But note that each thread is rotating concurrently

But What About Skiplists?

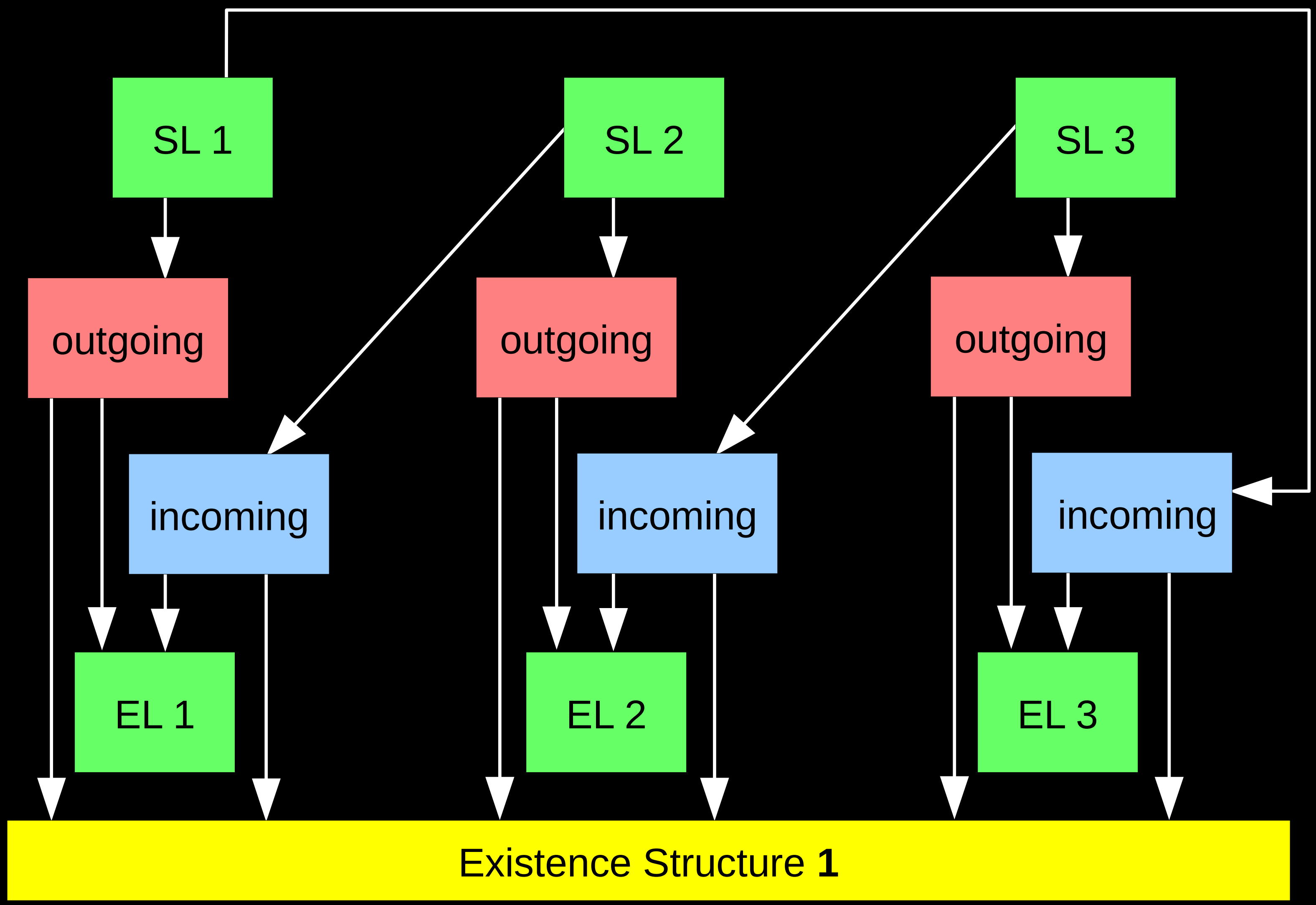
Rotate 3 Elements Through 3 Skiplists (1/4)



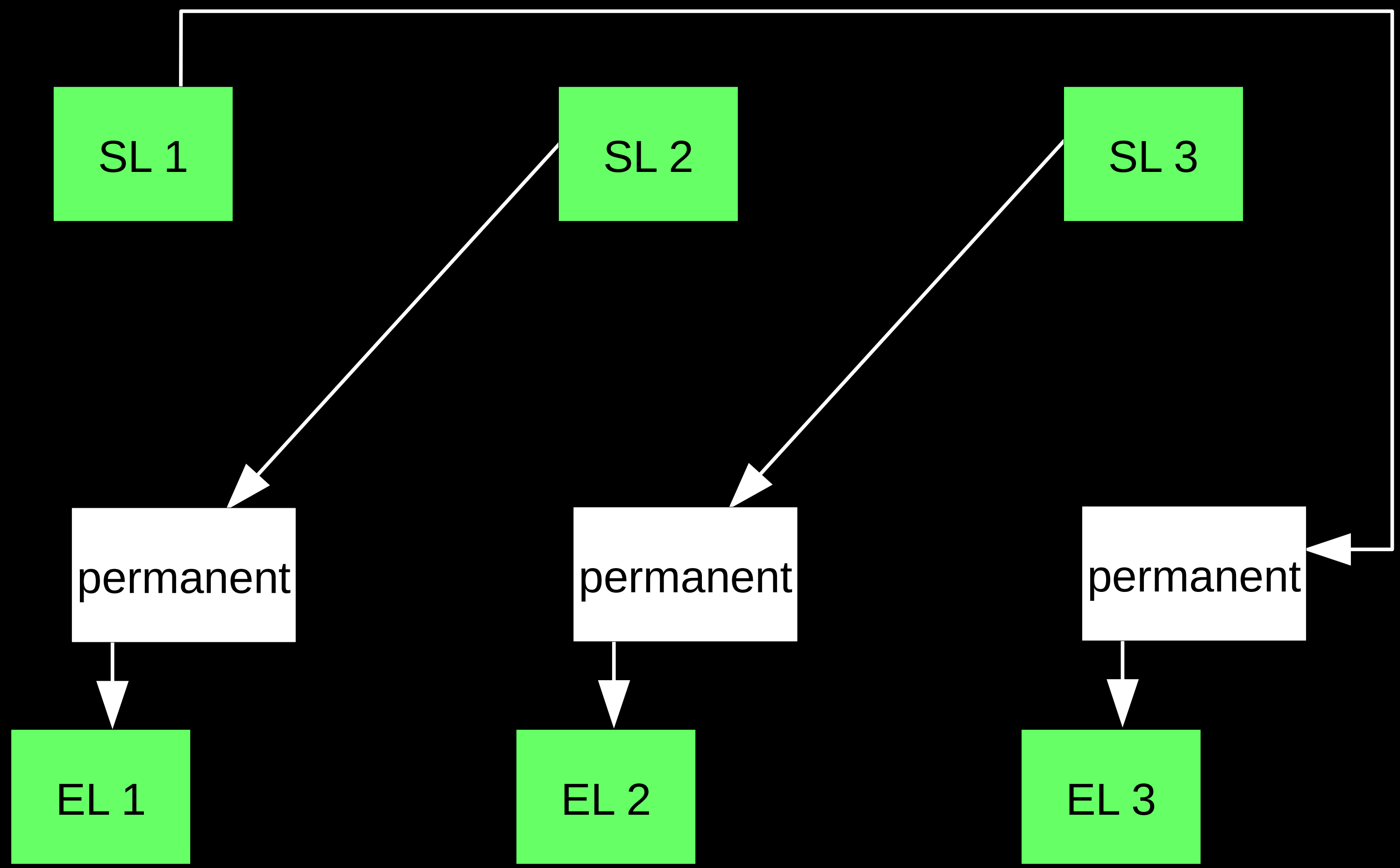
Rotate 3 Elements Through 3 Skiplists (2/4)



Rotate 3 Elements Through 3 Skiplists (3/4)



Rotate 3 Elements Through 3 Skiplists (4/4)



Data to Rotate 3 Elements Through 3 Skiplists

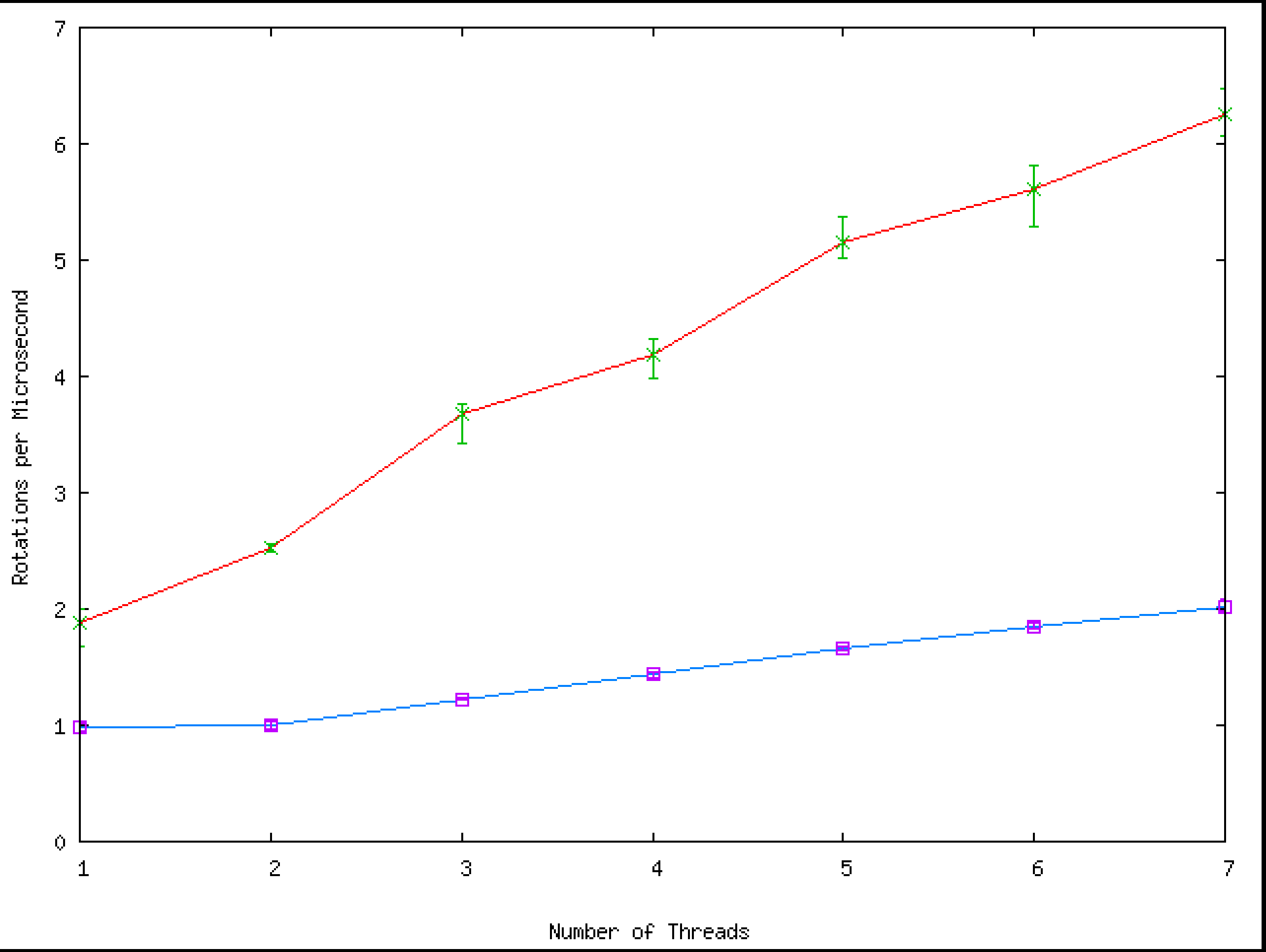
```
struct keyvalue {
        unsigned long key;
        unsigned long value;
        atomic_t refcnt;
};

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?



Hash table

Skiplist

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

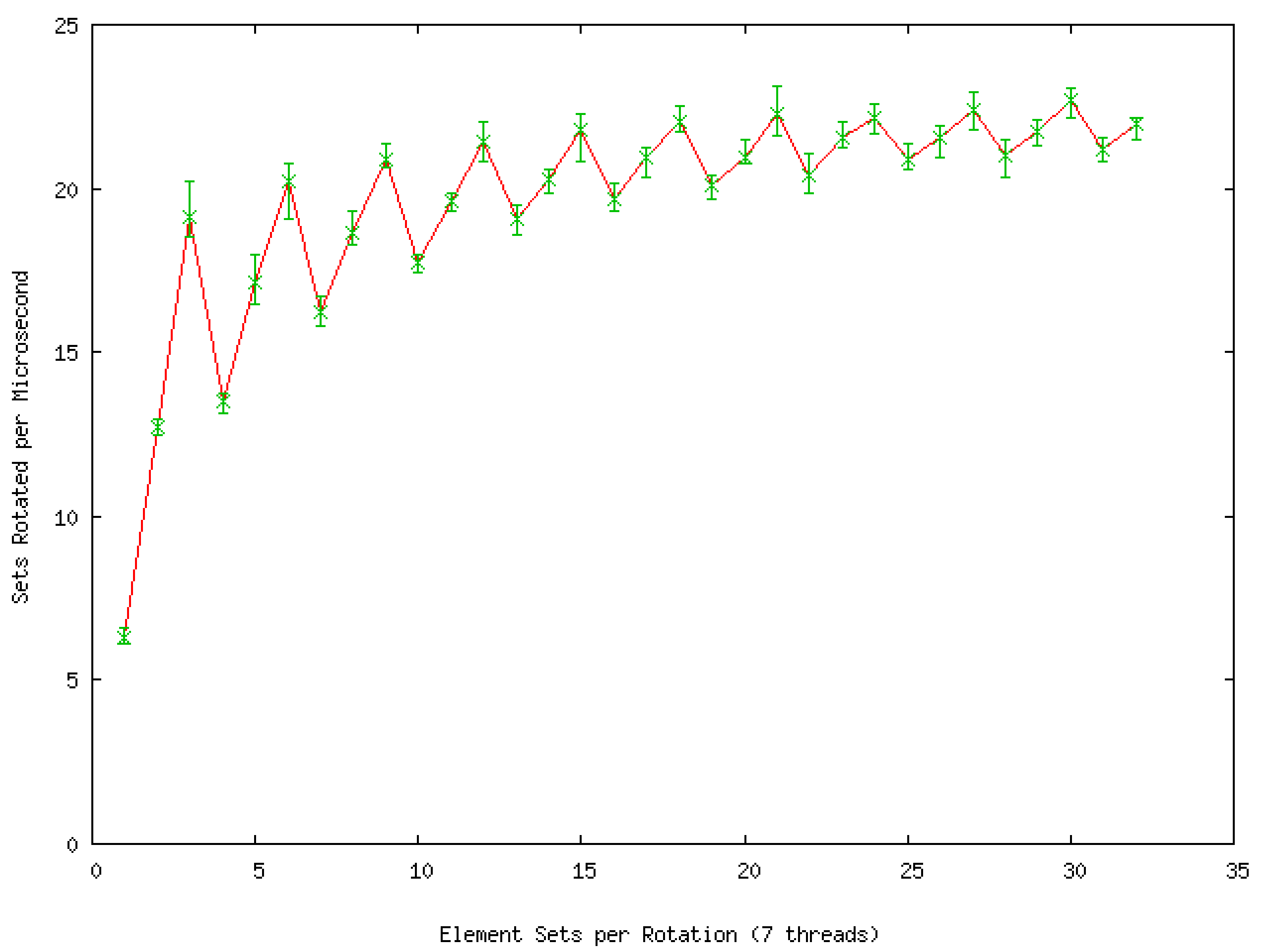
But Can We Atomically Rotate More Elements?

- 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???

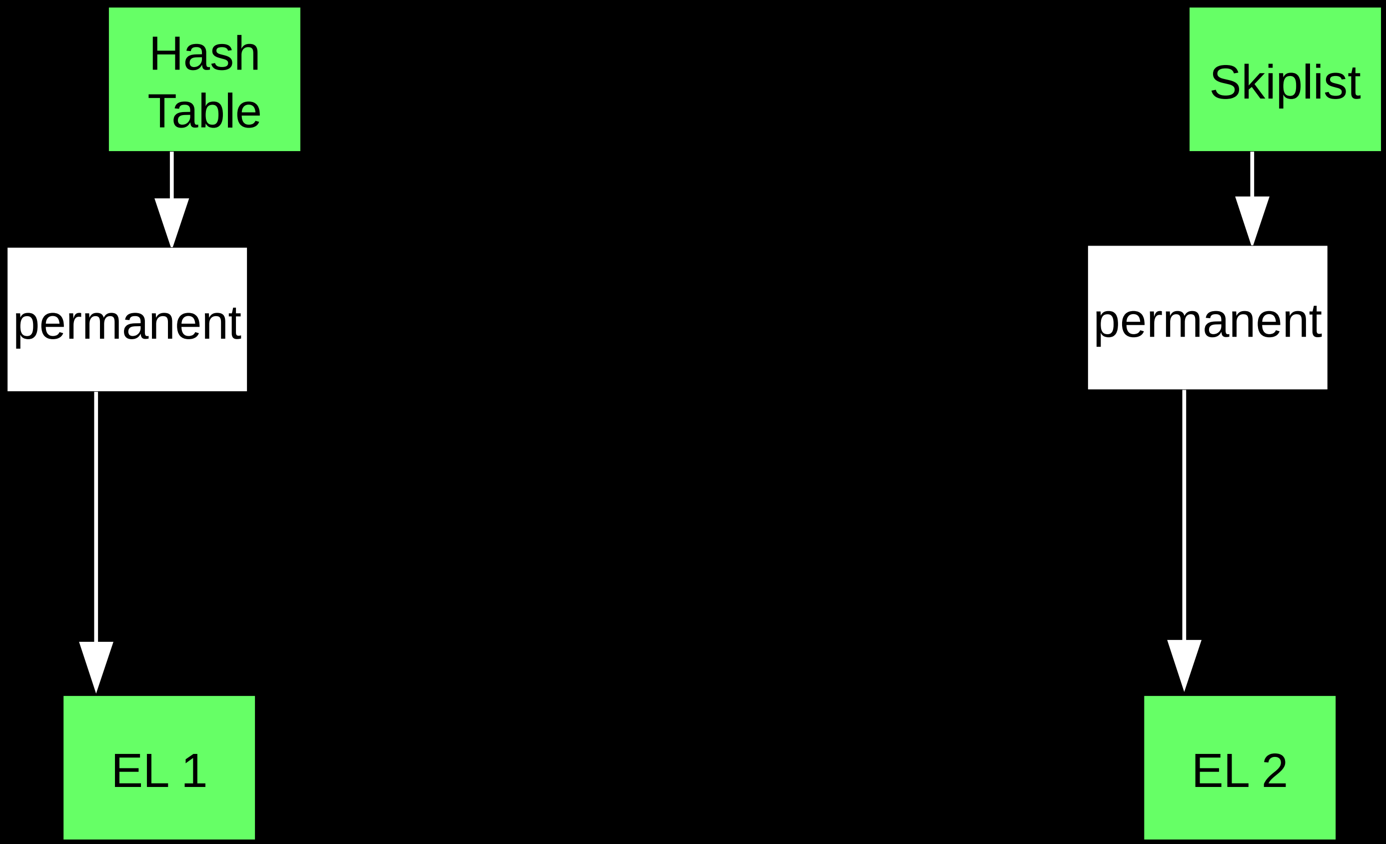
Even Bigger Mystery: Why Rotate This Way???

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

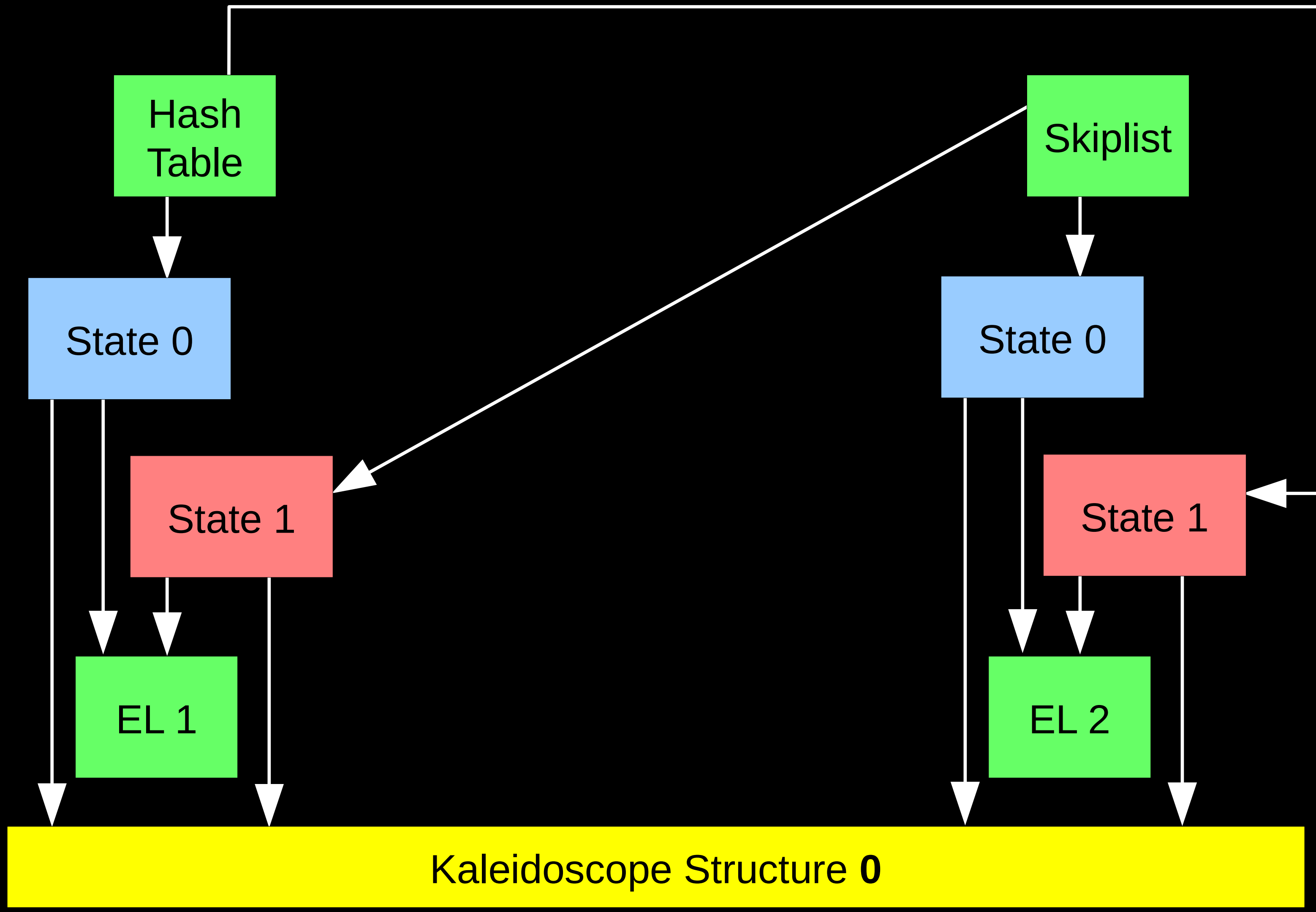
Even Bigger Mystery: Why Rotate This Way???

- 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

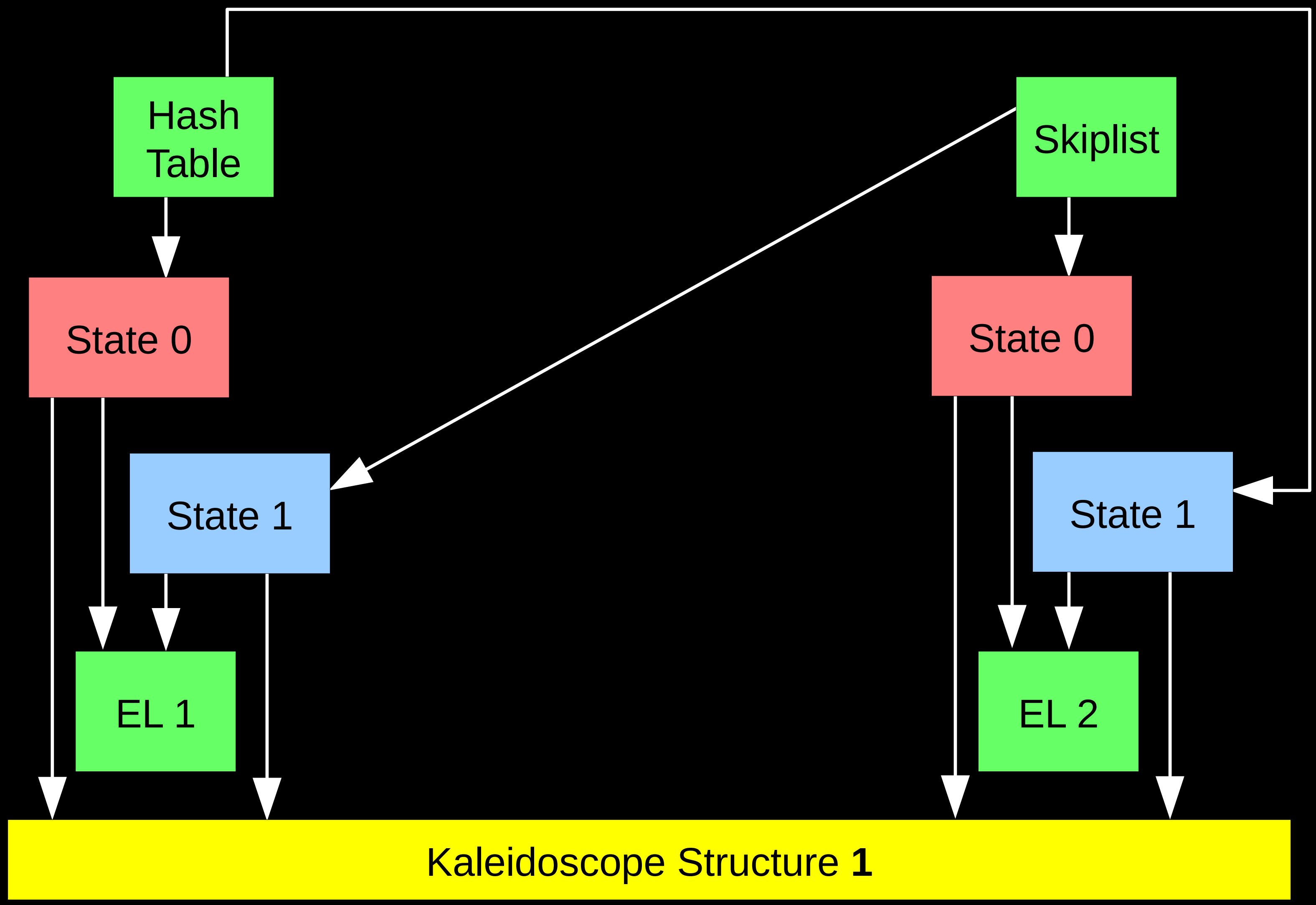
Rotate Through Hash Table & Skiplist (1/3)



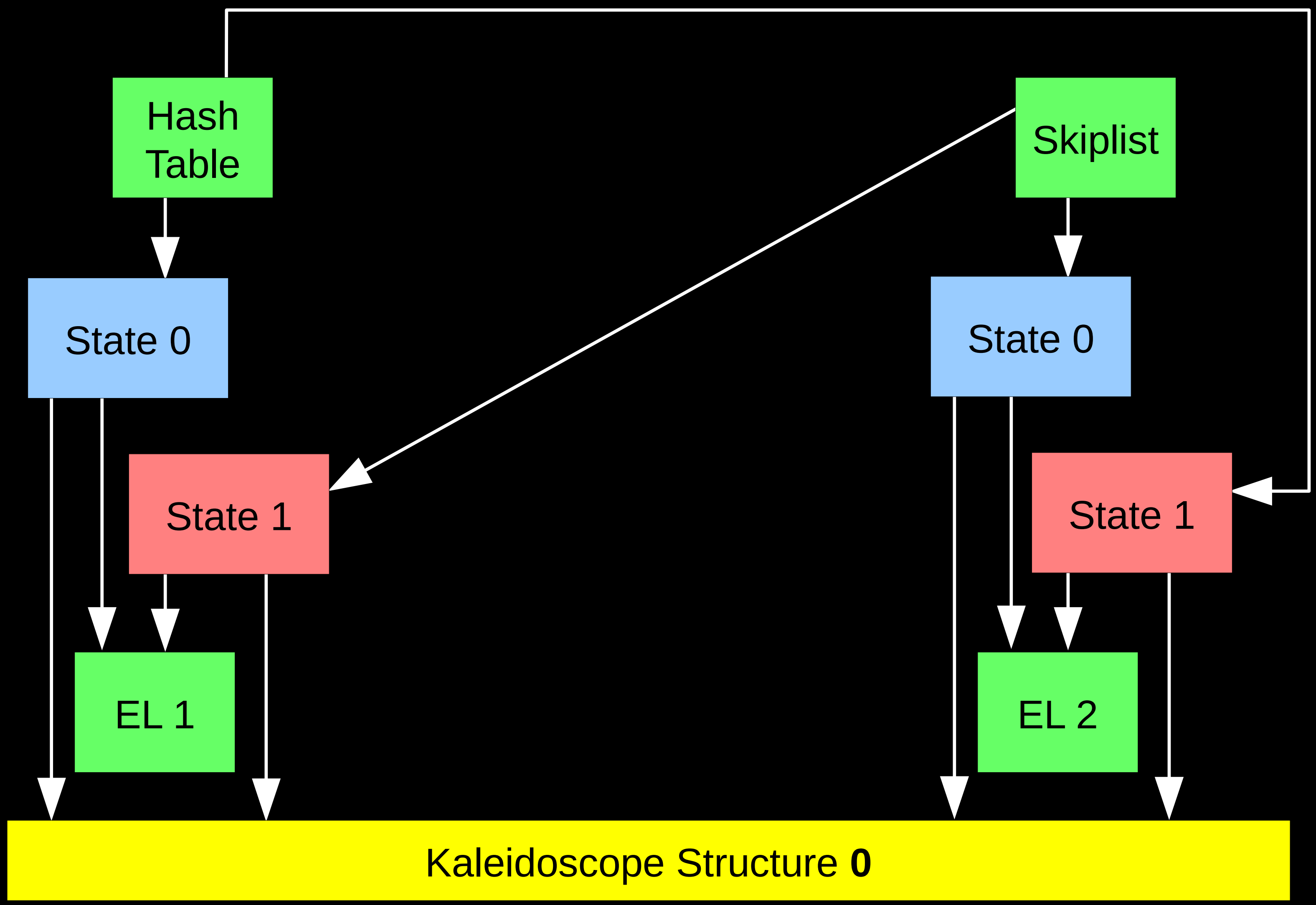
Rotate Through Hash Table & Skiplist (2/3)



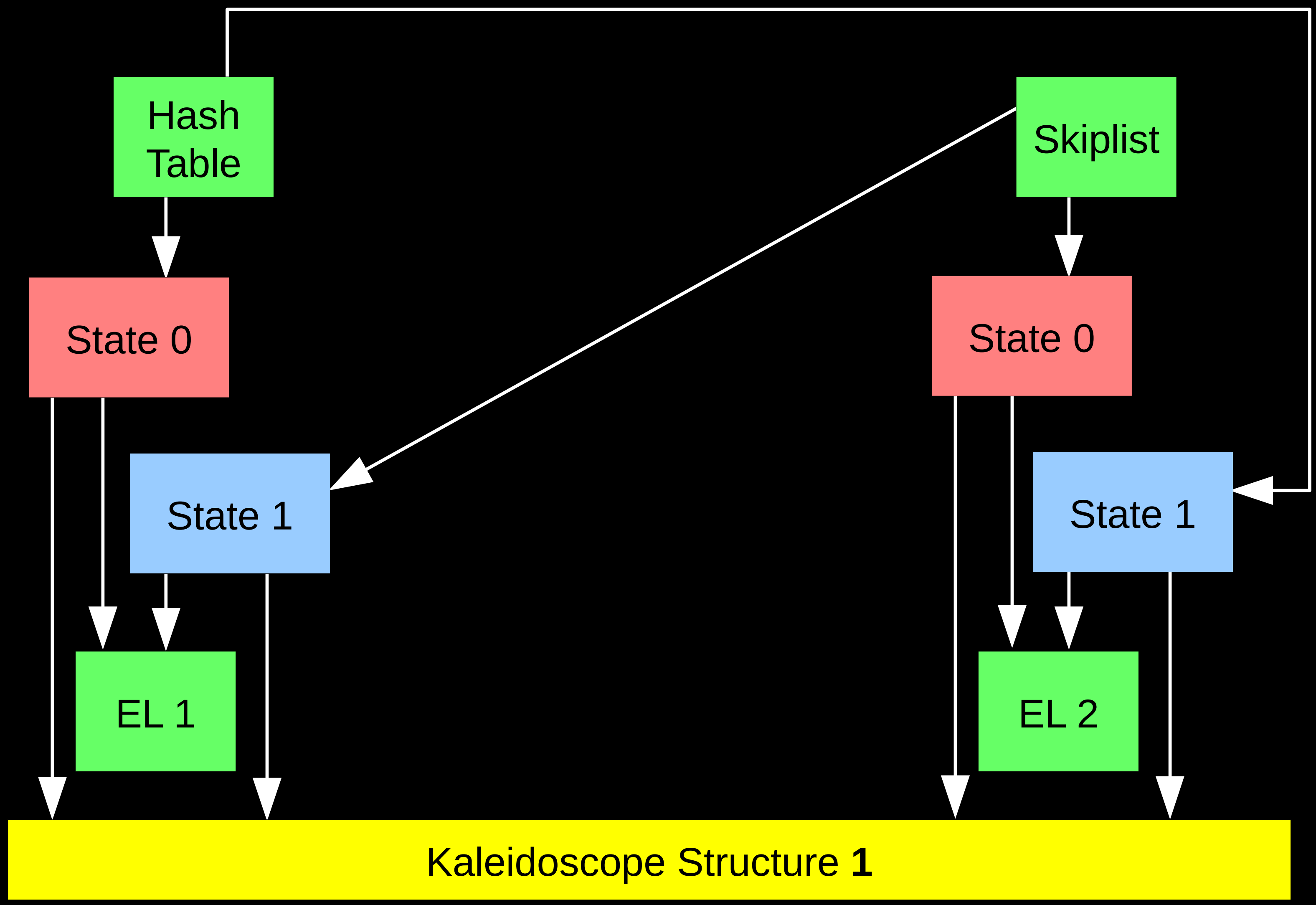
Rotate Through Hash Table & Skiplist (3/3)



Rotate Through Hash Table & Skiplist (2/3)



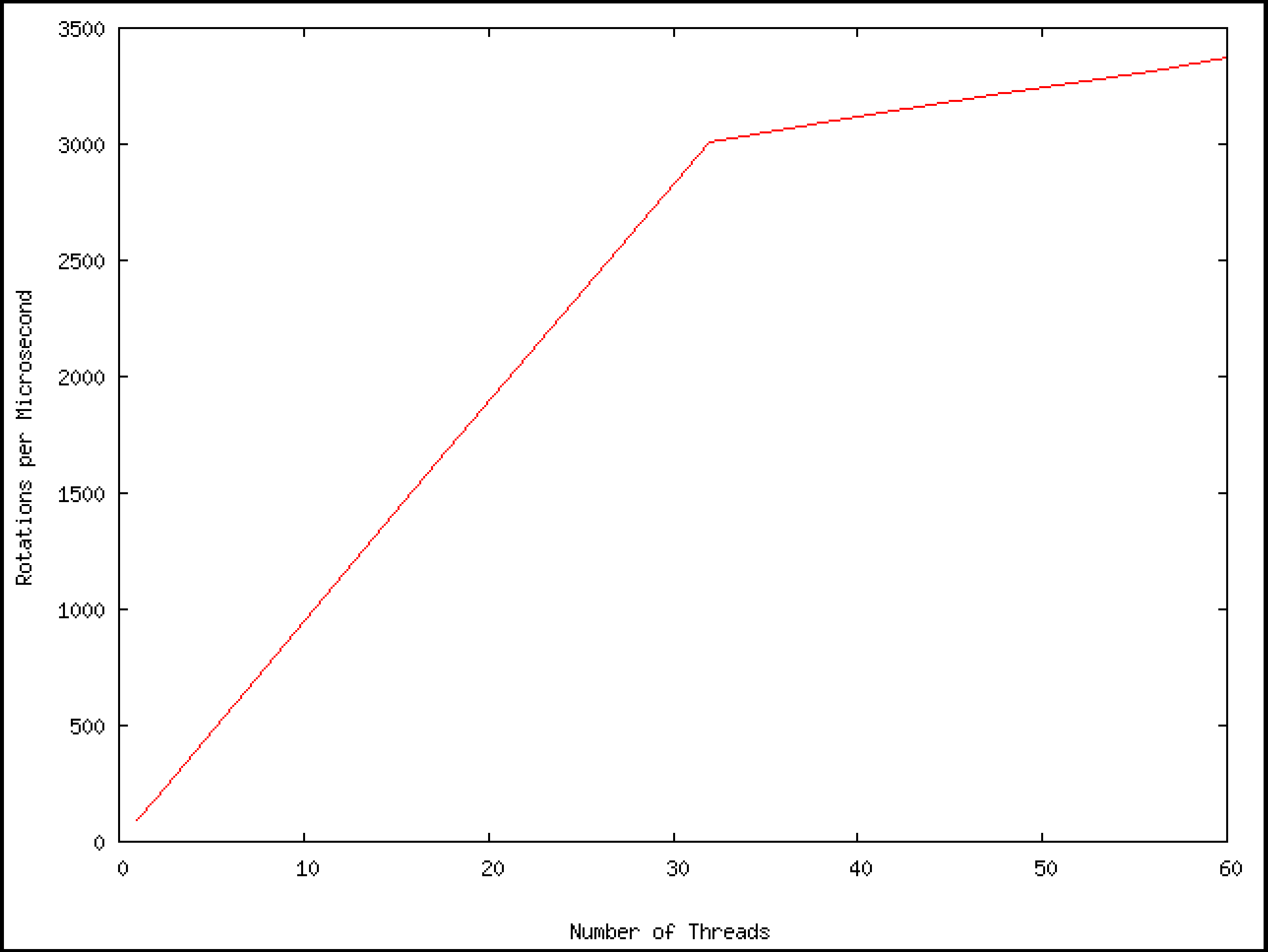
Rotate Through Hash Table & Skiplist (3/3)



Very Tight Loop...

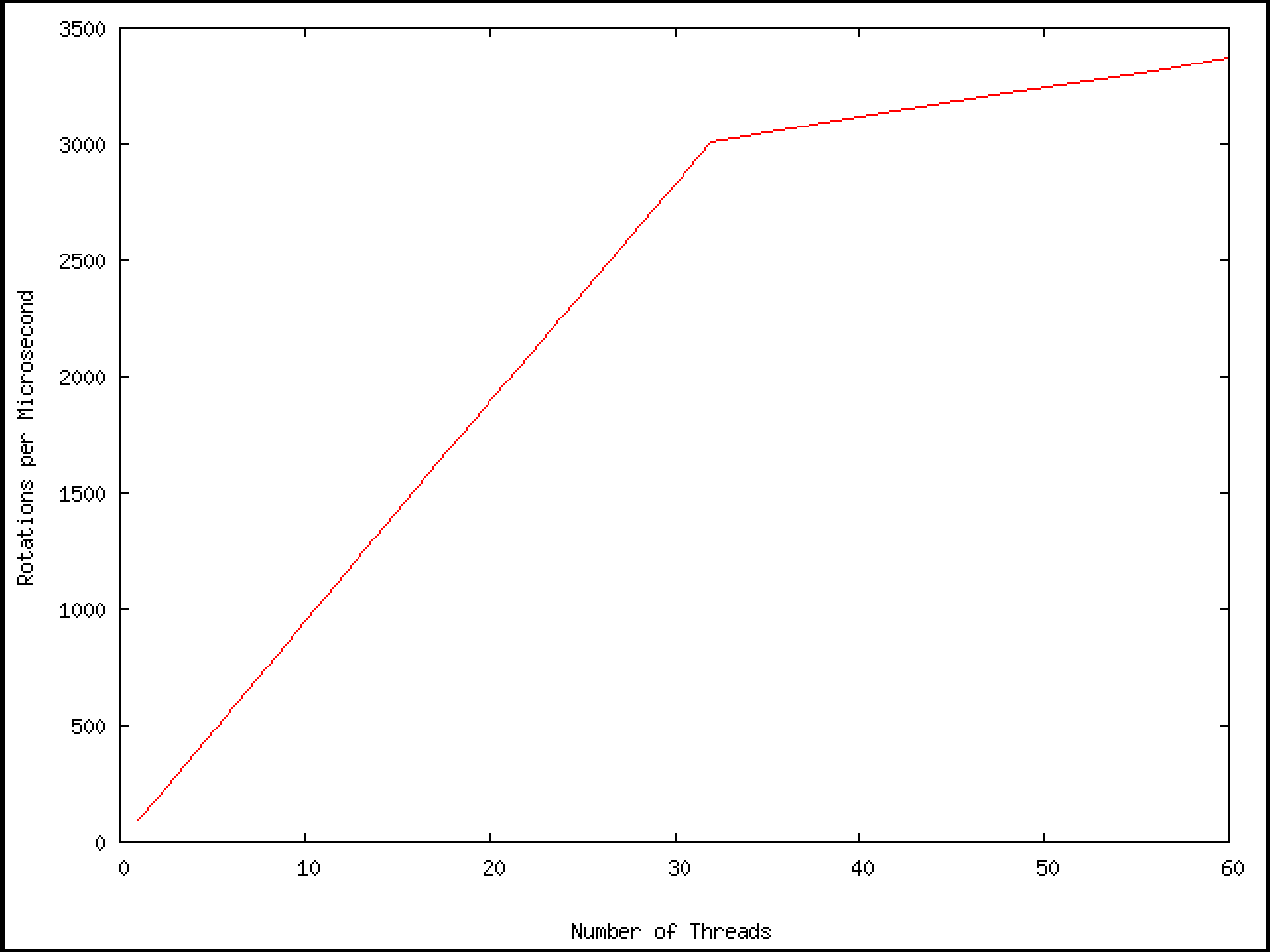
```
while (ACCESS_ONCE(goflag) == GOFLAG_RUN) {
        kaleidoscope_set_state(kgp, nrotations % 2);
        nrotations++;
}
```

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!!!

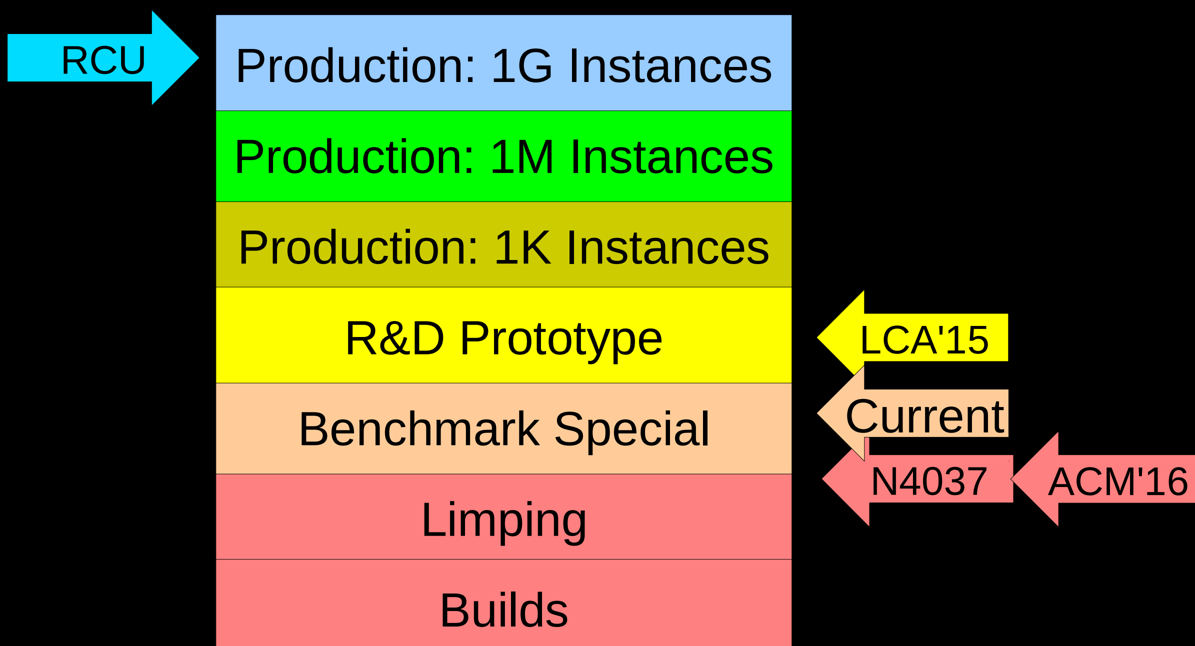
When Might You Use Existence-Based Update?

- 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

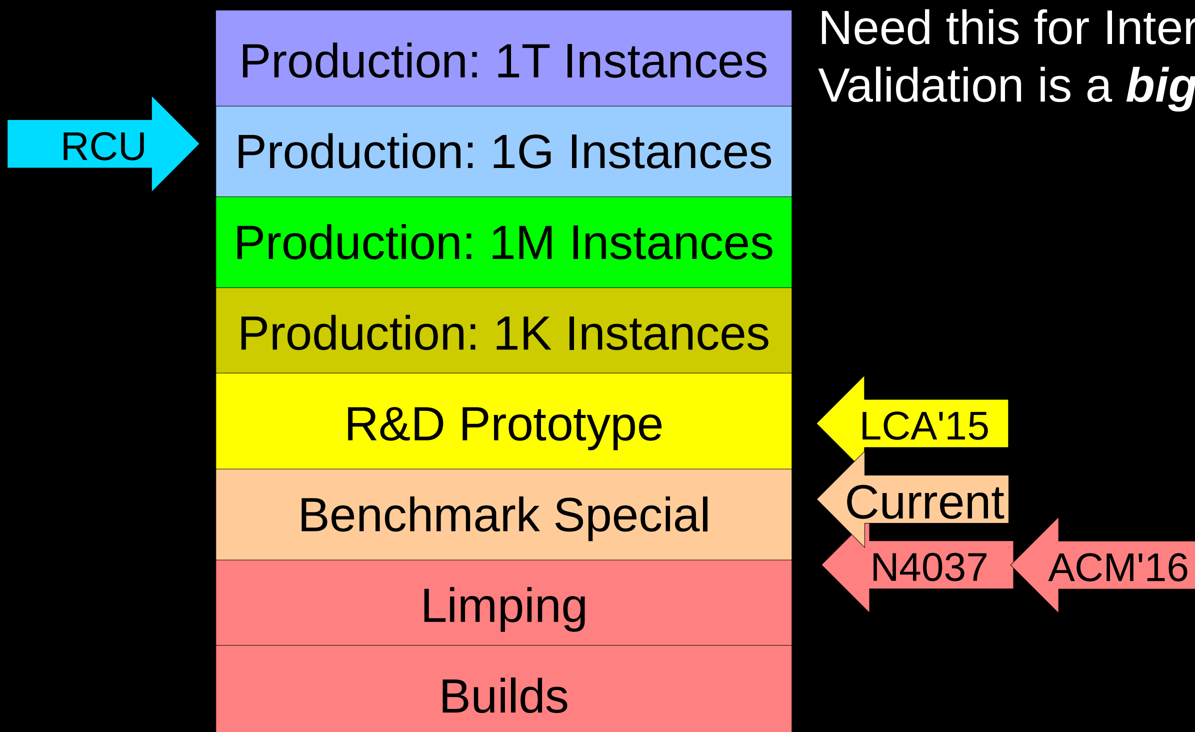
- No, it is ***not*** production ready (but was getting there)



Existence Structures: Production Readiness

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

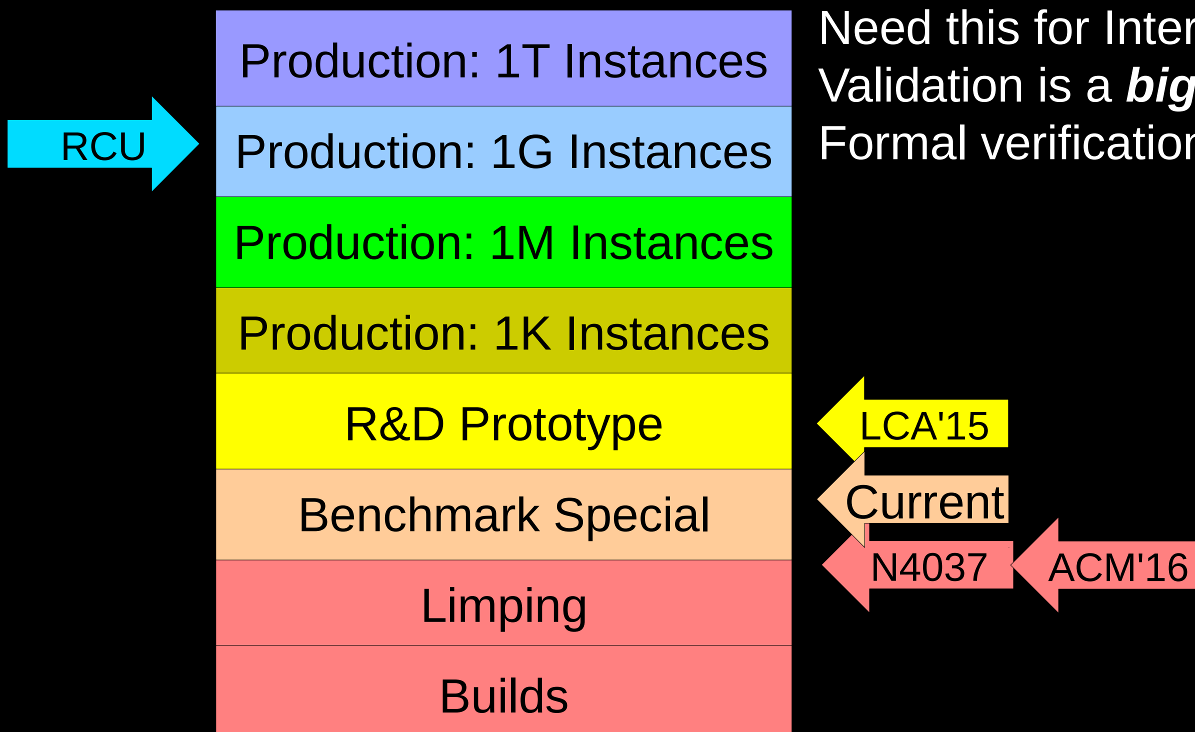
Need this for Internet of Things,
Validation is a ***big*** unsolved problem



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

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!

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.

Questions?

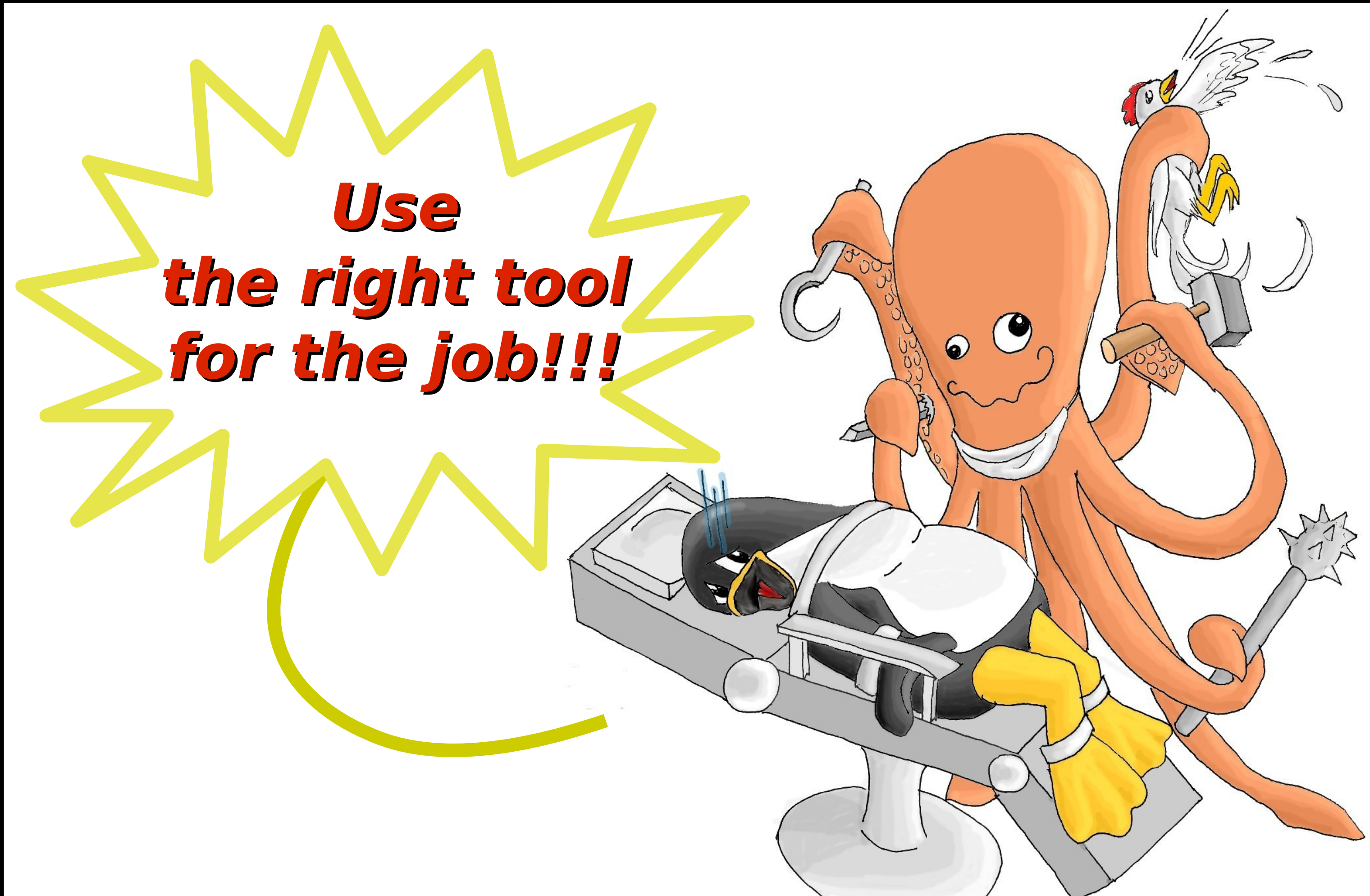


Image copyright © 2004 Melissa McKenney