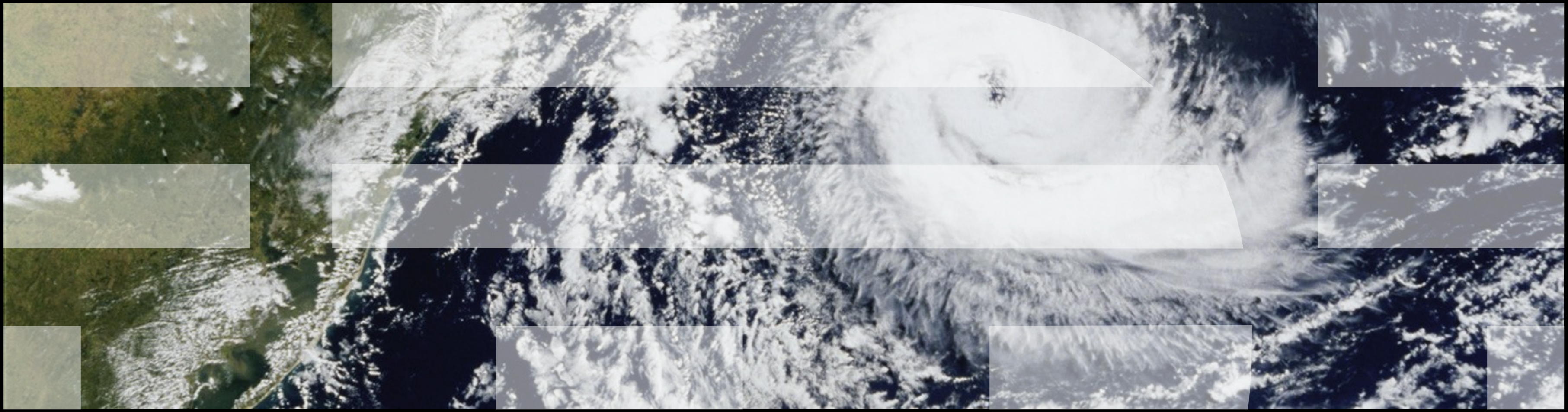
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology

TU-Dresden Distributed Operating Systems, June 1, 2015

High-Performance and Scalable Updates: The Issaquah Challenge



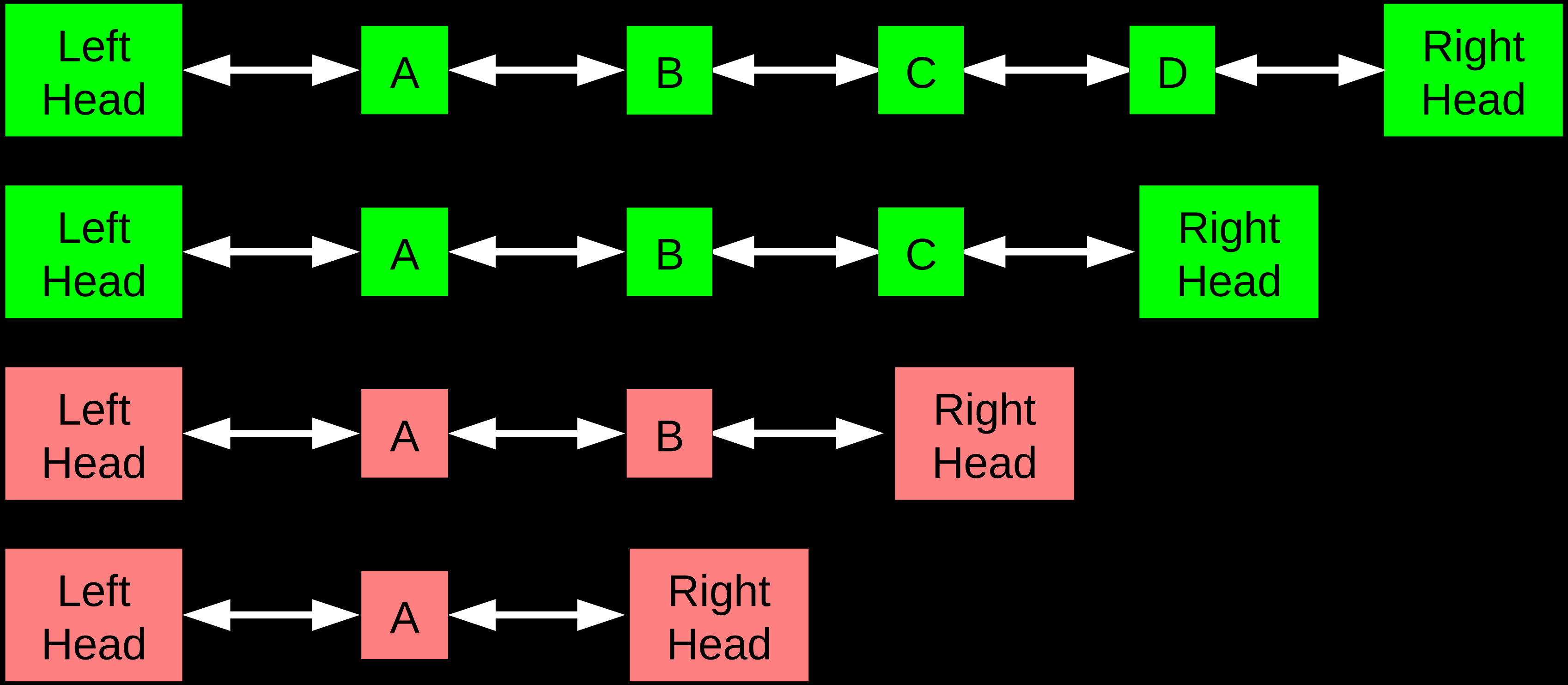
Overview

- Before the Issaquah Challenge
- The Issaquah Challenge
- Aren't parallel updates a solved problem?
- Special case for parallel updates
  - Per-CPU/thread processing
  - Read-only traversal to location being updated
  - Existence-based updates
- The Issaquah Challenge: One Solution

Before the Issaquah Challenge

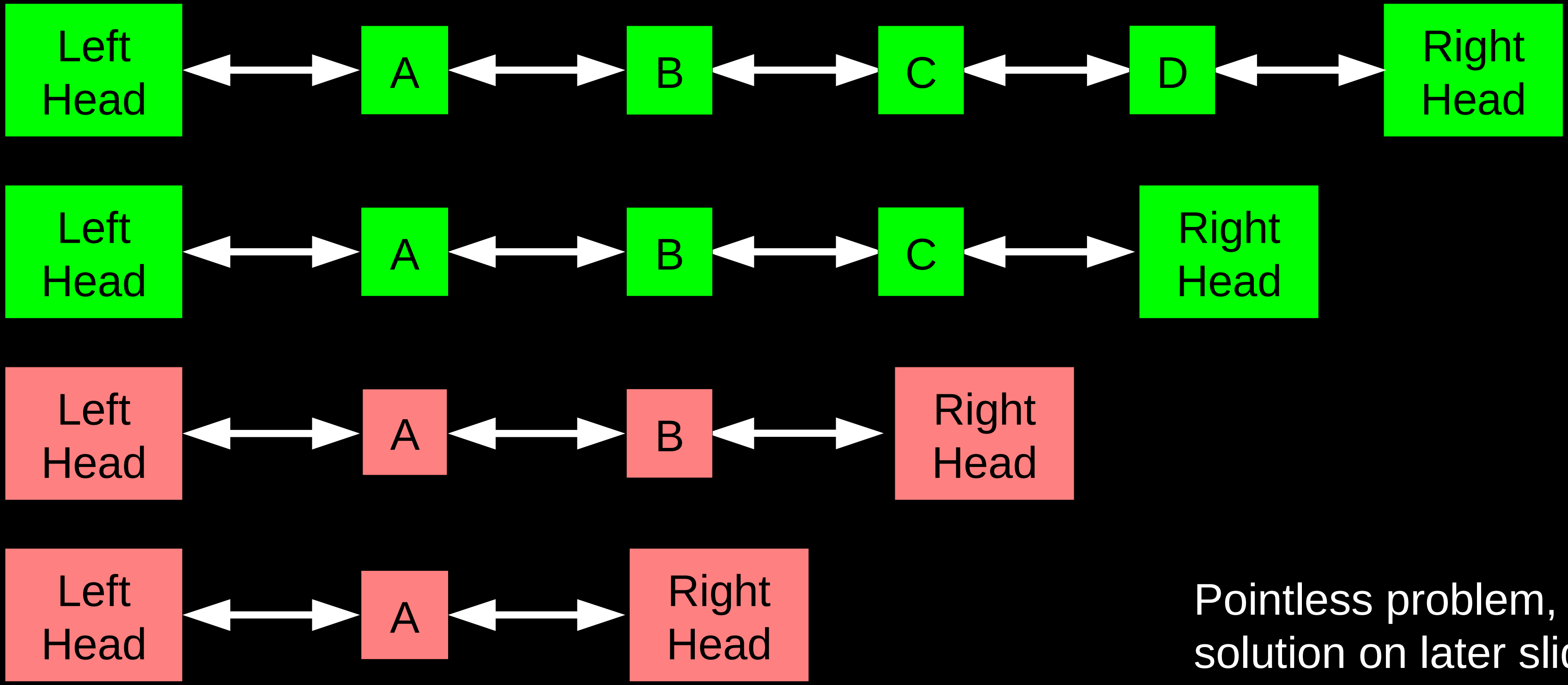
Before the Issaquah Challenge: Double-Ended Queue

- 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



Before the Issaquah Challenge: Double-Ended Queue

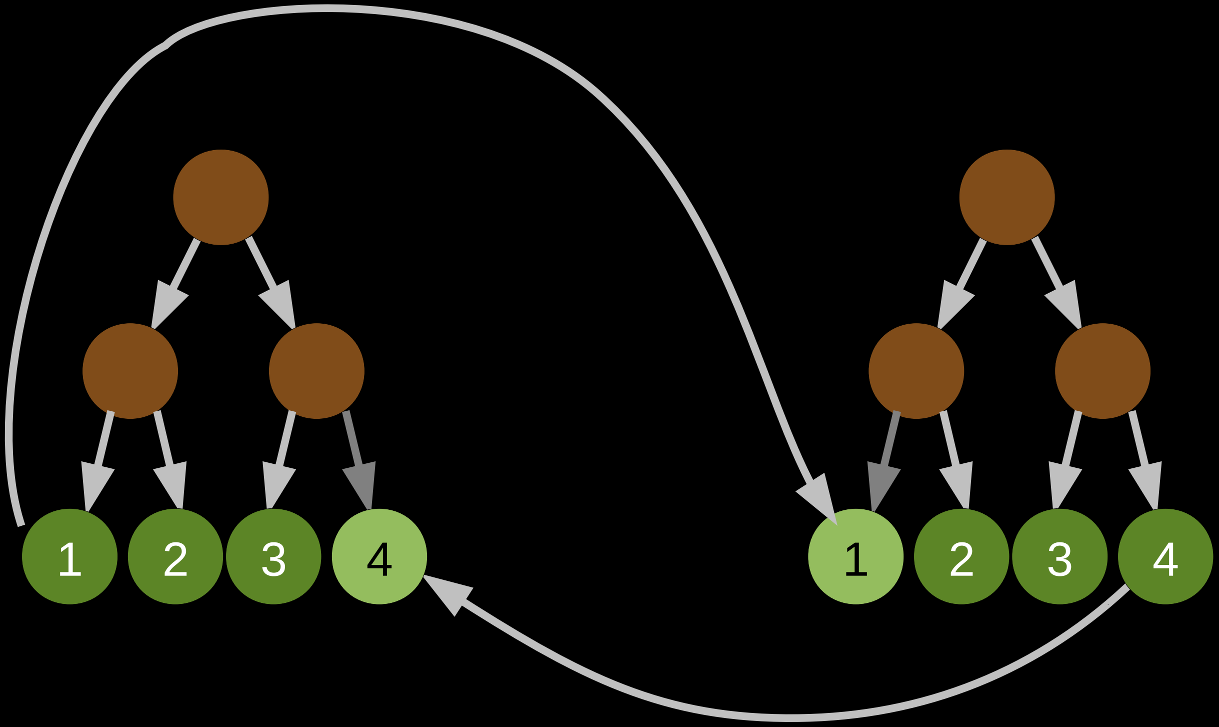
- Can you create a trivial lock-based deque allowing concurrent pushes and pops at both ends?
  - Coordination required if the deque contains only one or two elements
  - But coordination is not required for three or more elements



Pointless problem, but solution on later slide...

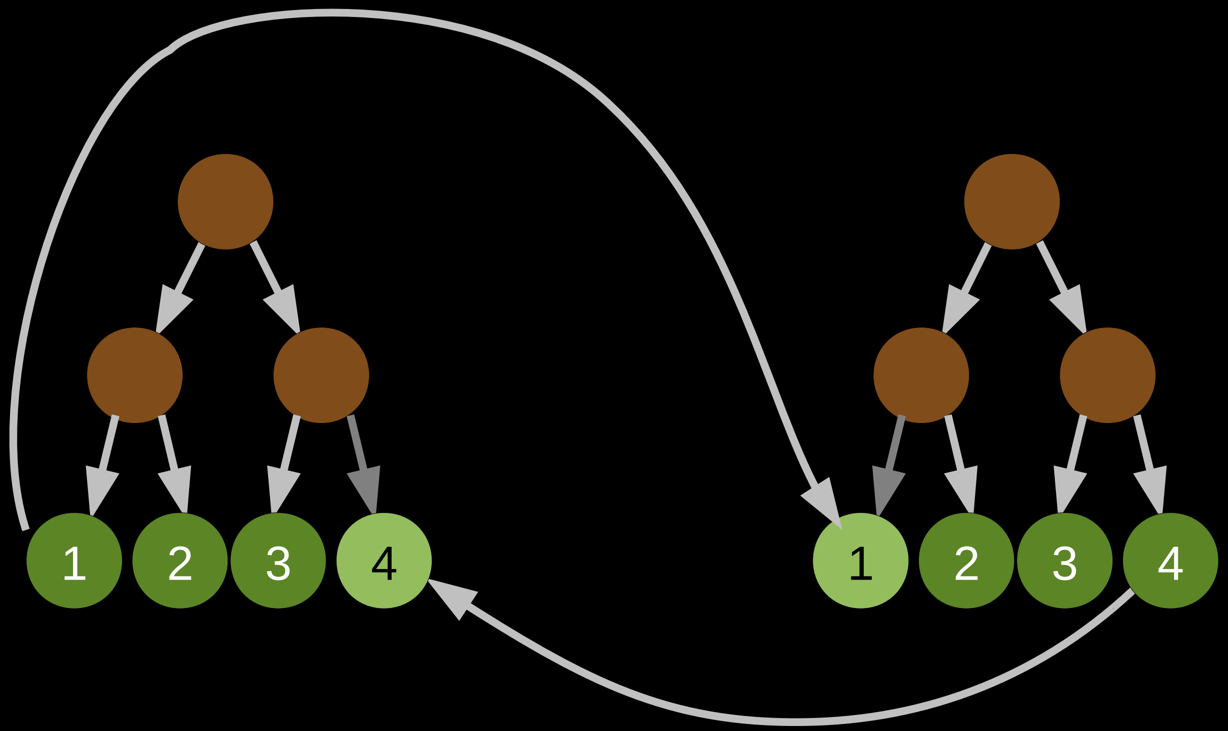
Atomic Multi-Structure Update: Issaquah Challenge

Atomic Multi-Structure Update: Issaquah Challenge



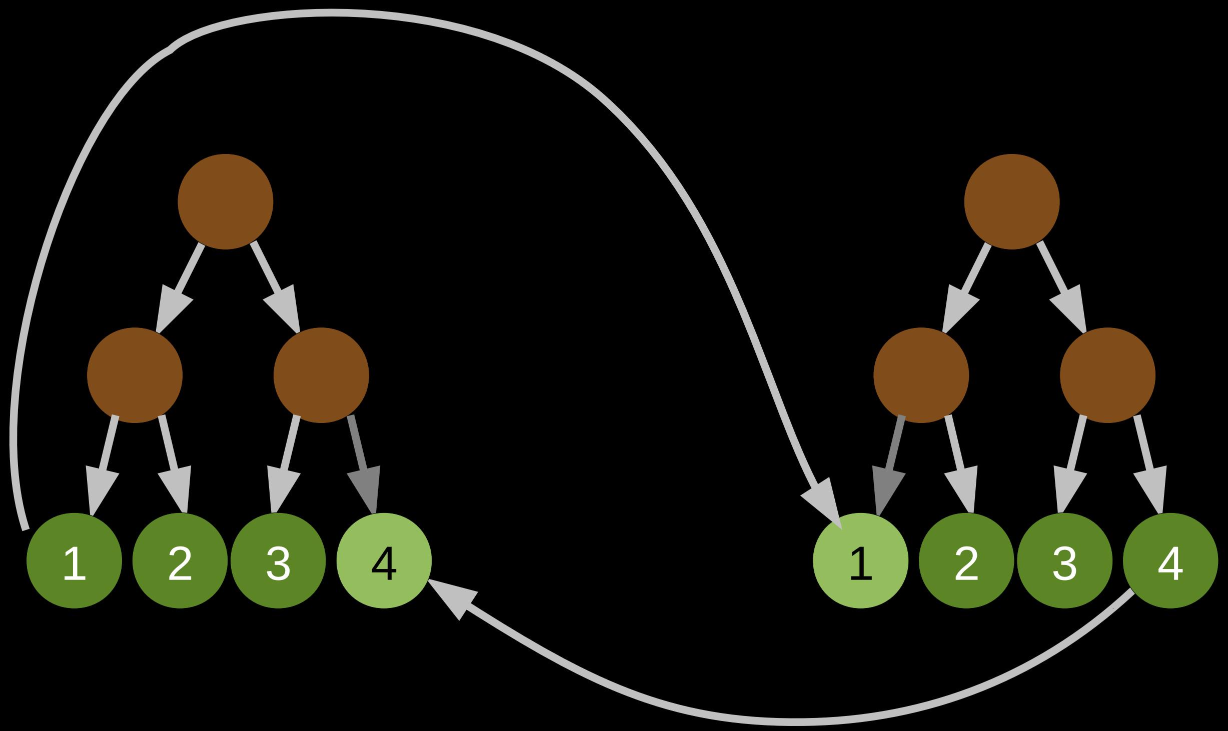
Atomically move element 1 from left to right tree
Atomically move element 4 from right to left tree

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!

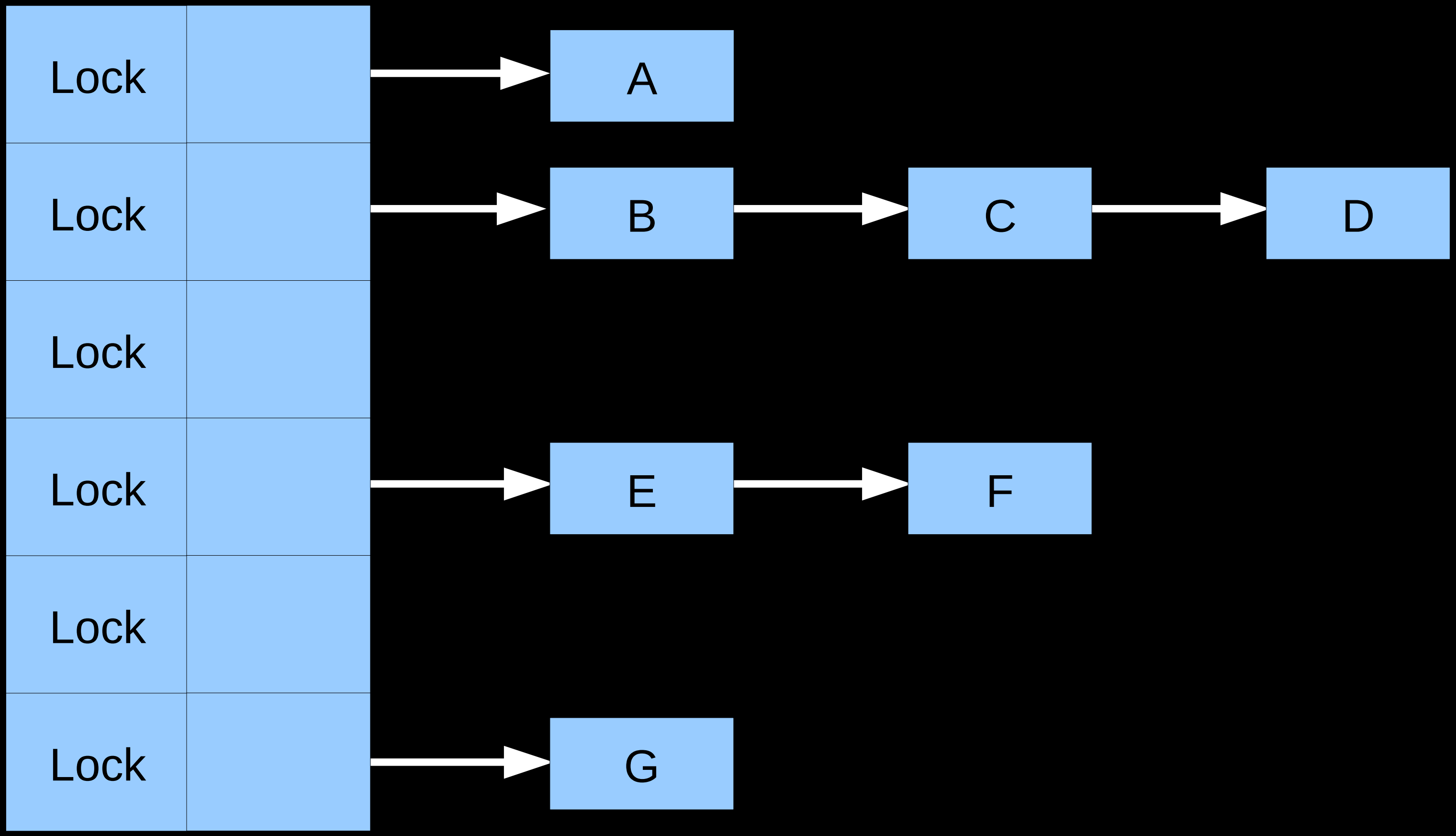
Atomic Multi-Structure Update: Issaquah Challenge



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”

But Aren't Parallel Updates A Solved Problem?

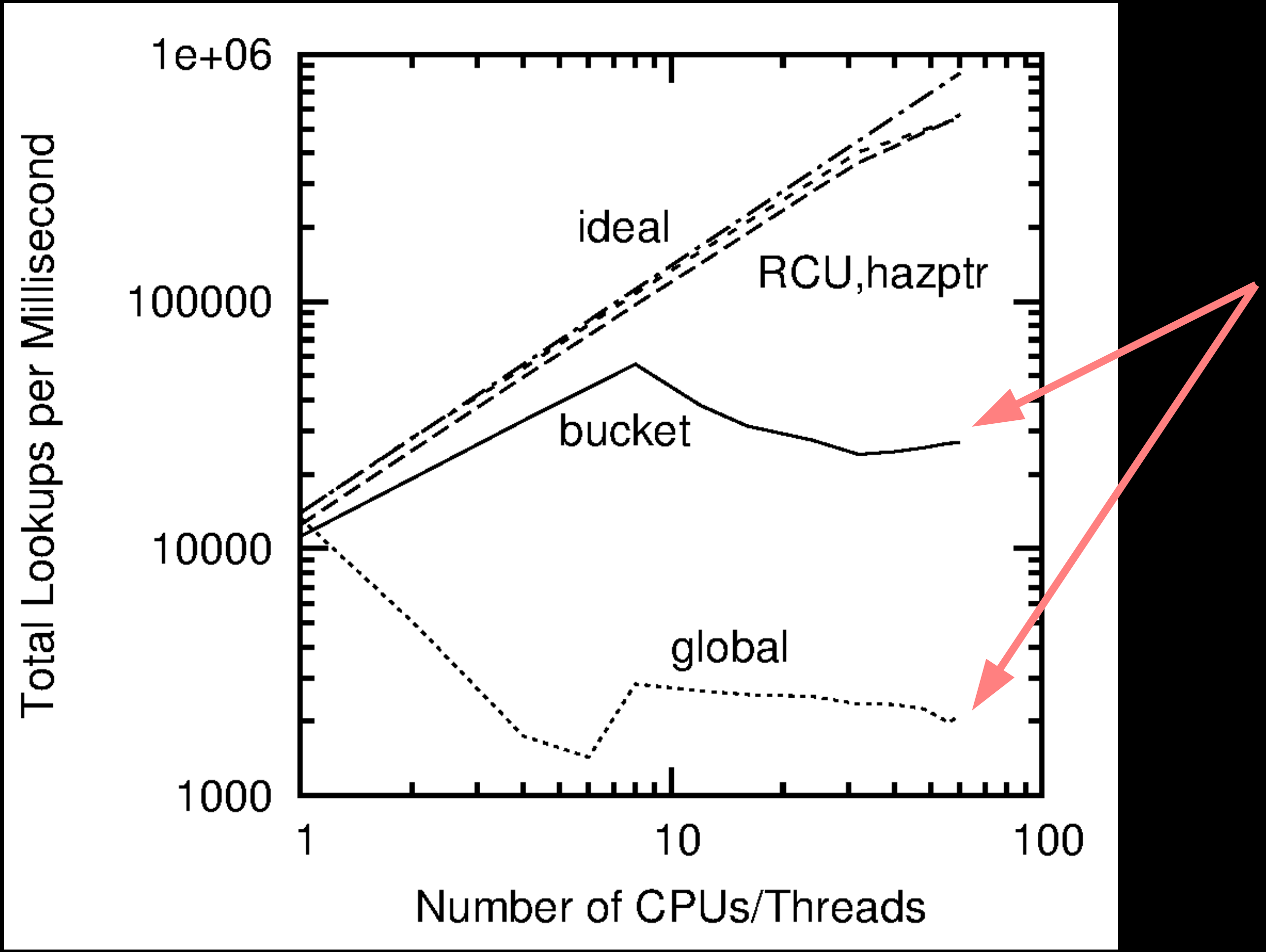
Parallel-Processing Workhorse: Hash Tables



Perfect partitioning leads to perfect performance and stunning scalability!

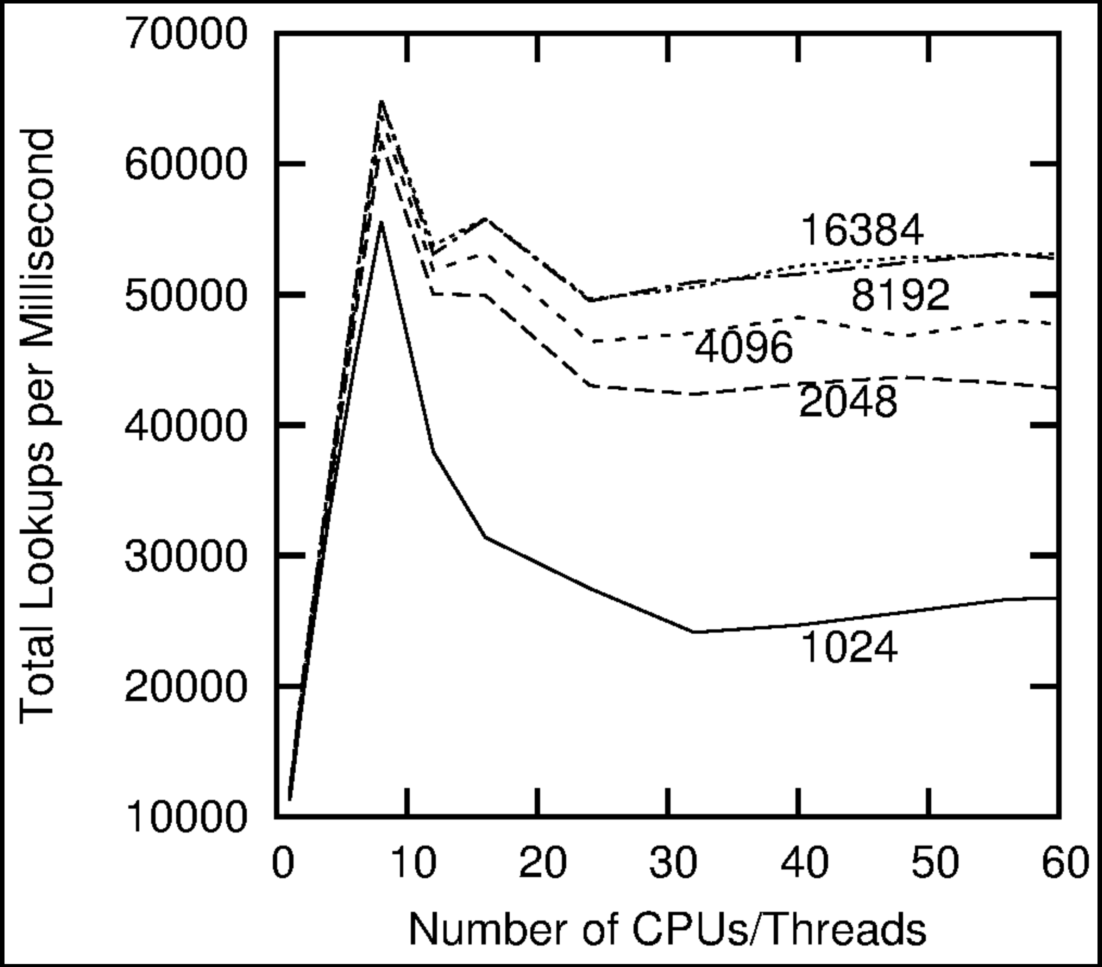
In theory, anyway...

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



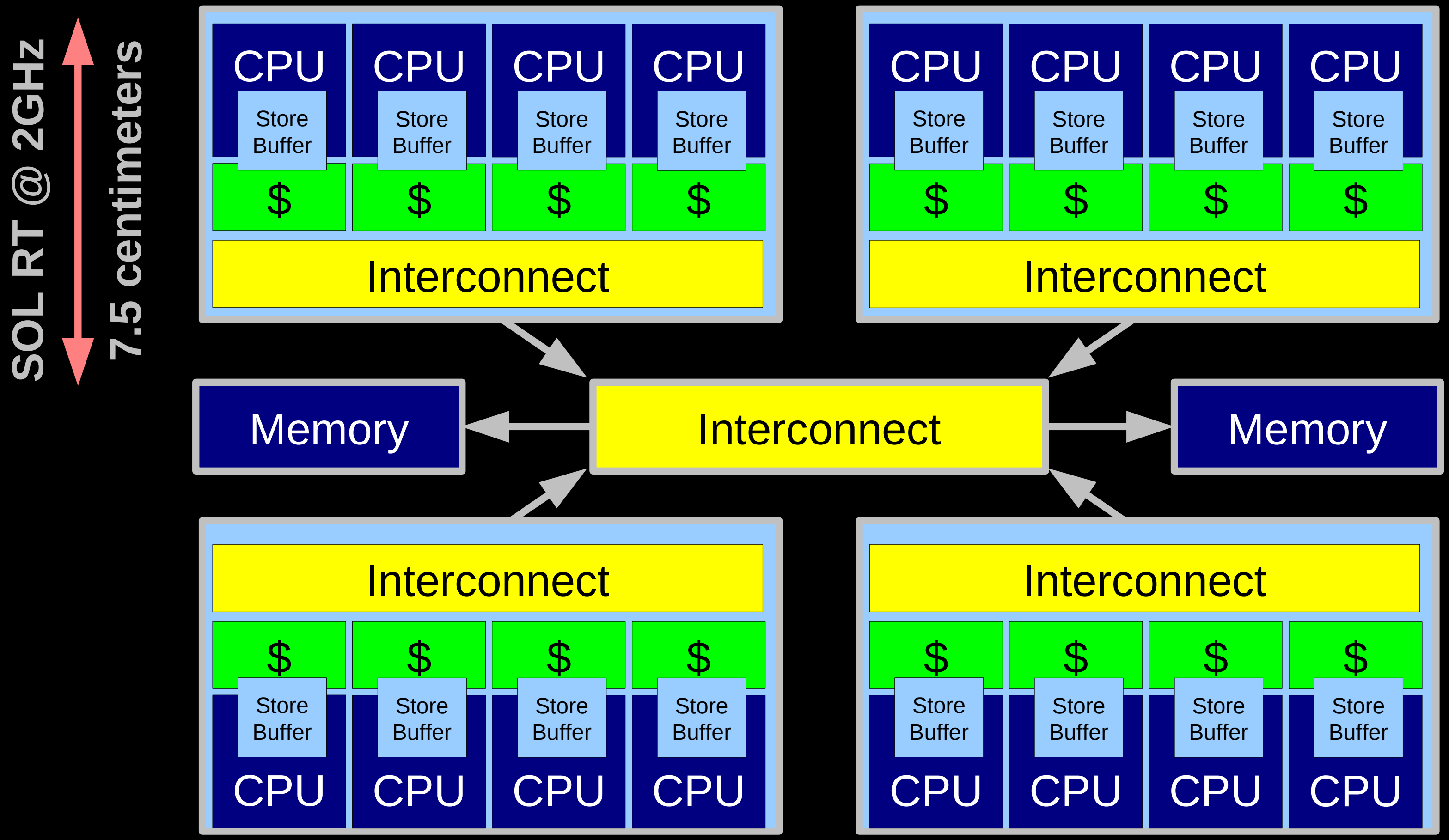
And the horrible thing? Updates are all locking ops!

But Hash Tables Are Partitionable! # of Buckets?



Some improvement, but...

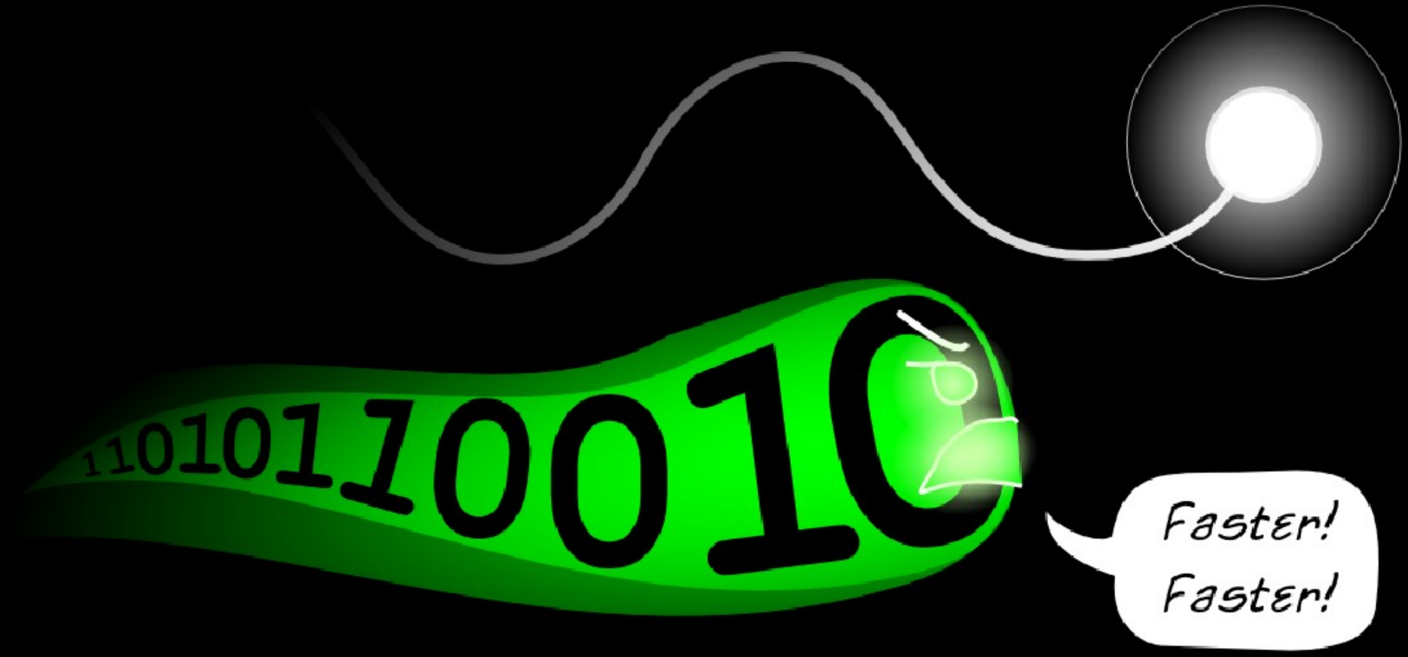
Hardware Structure and Laws of Physics



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

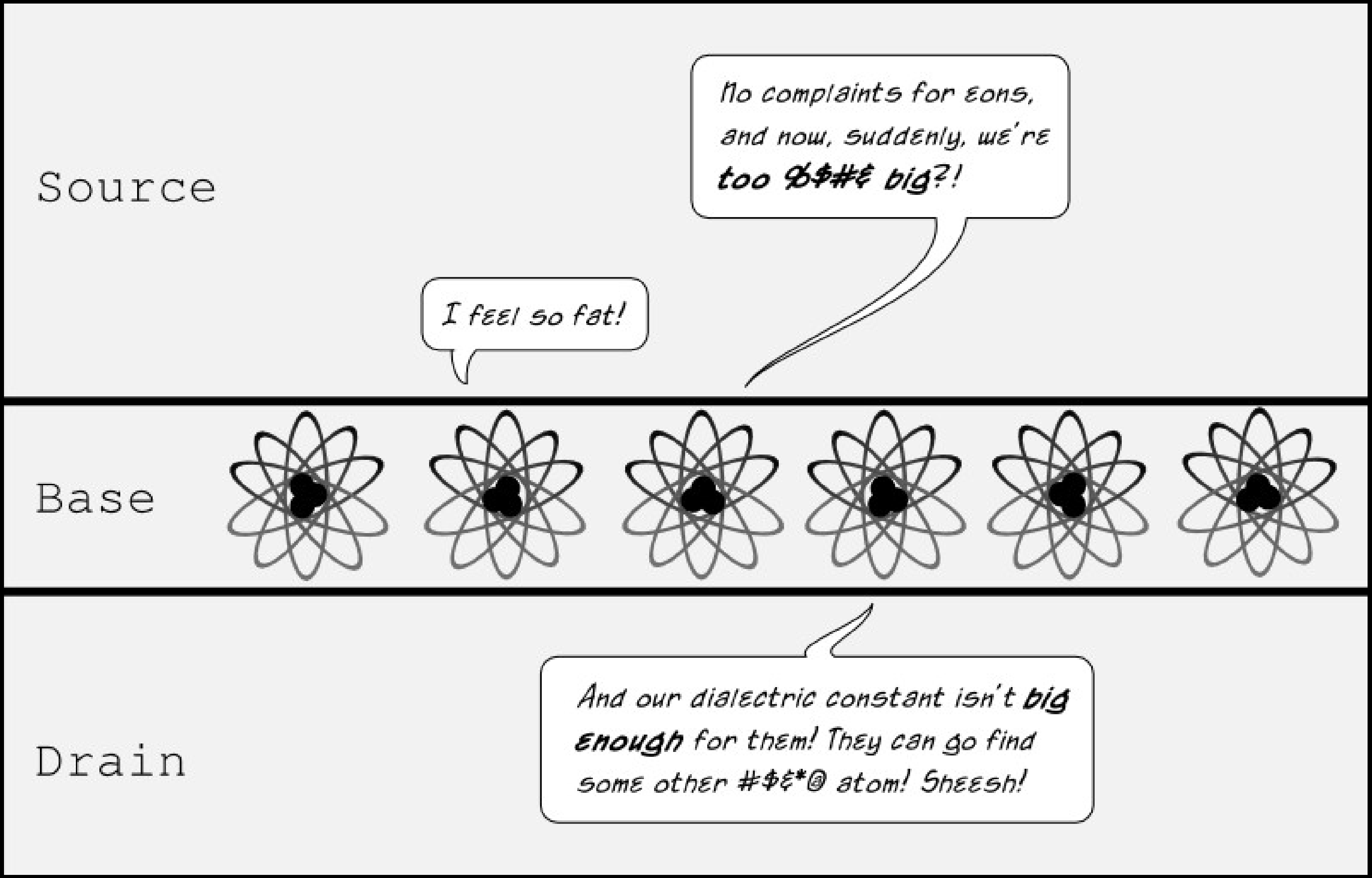
Two Problems With Fundamental Physics...

Problem With Physics #1: Finite Speed of Light



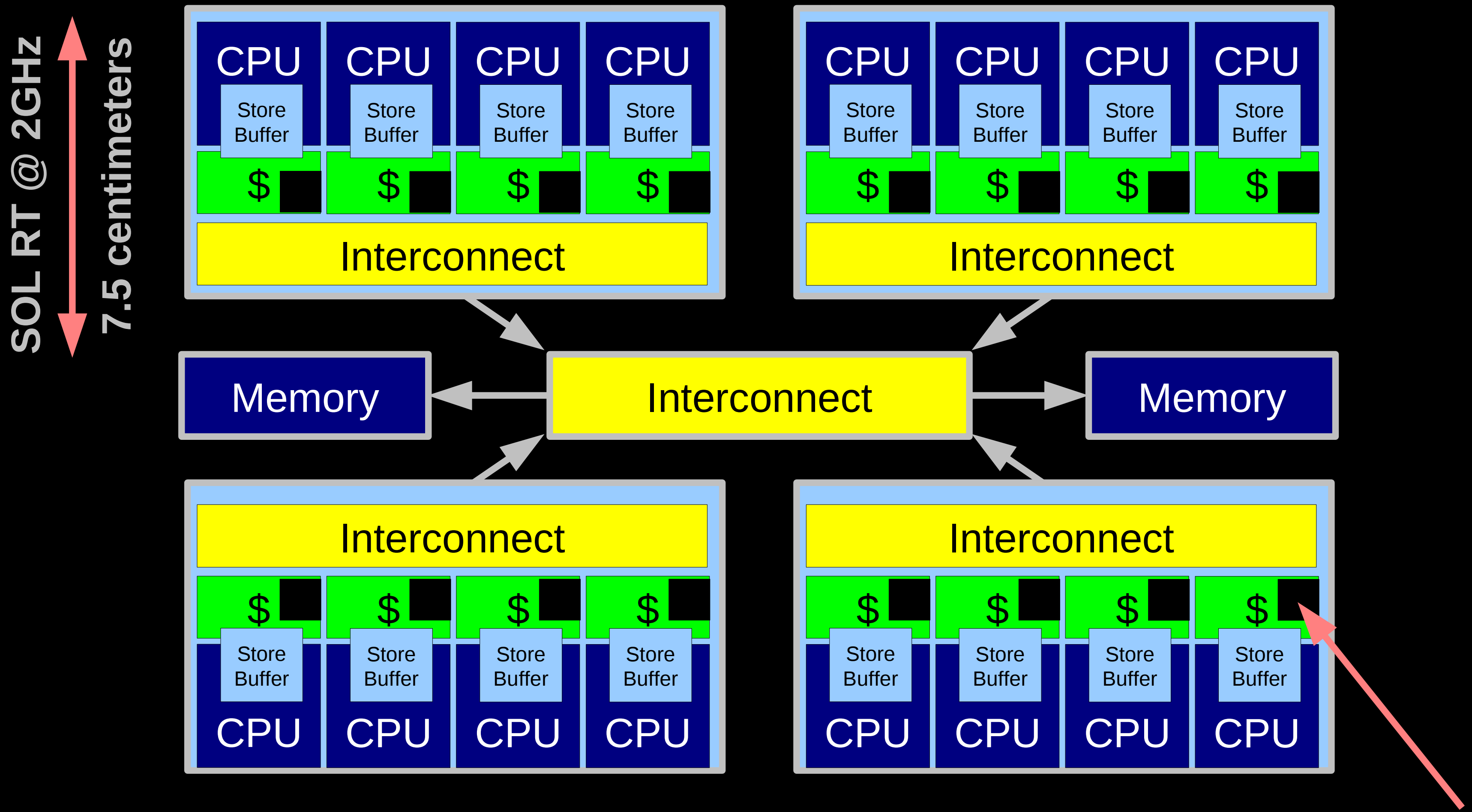
Observation by Stephen Hawking

Problem With Physics #2: Atomic Nature of Matter


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!

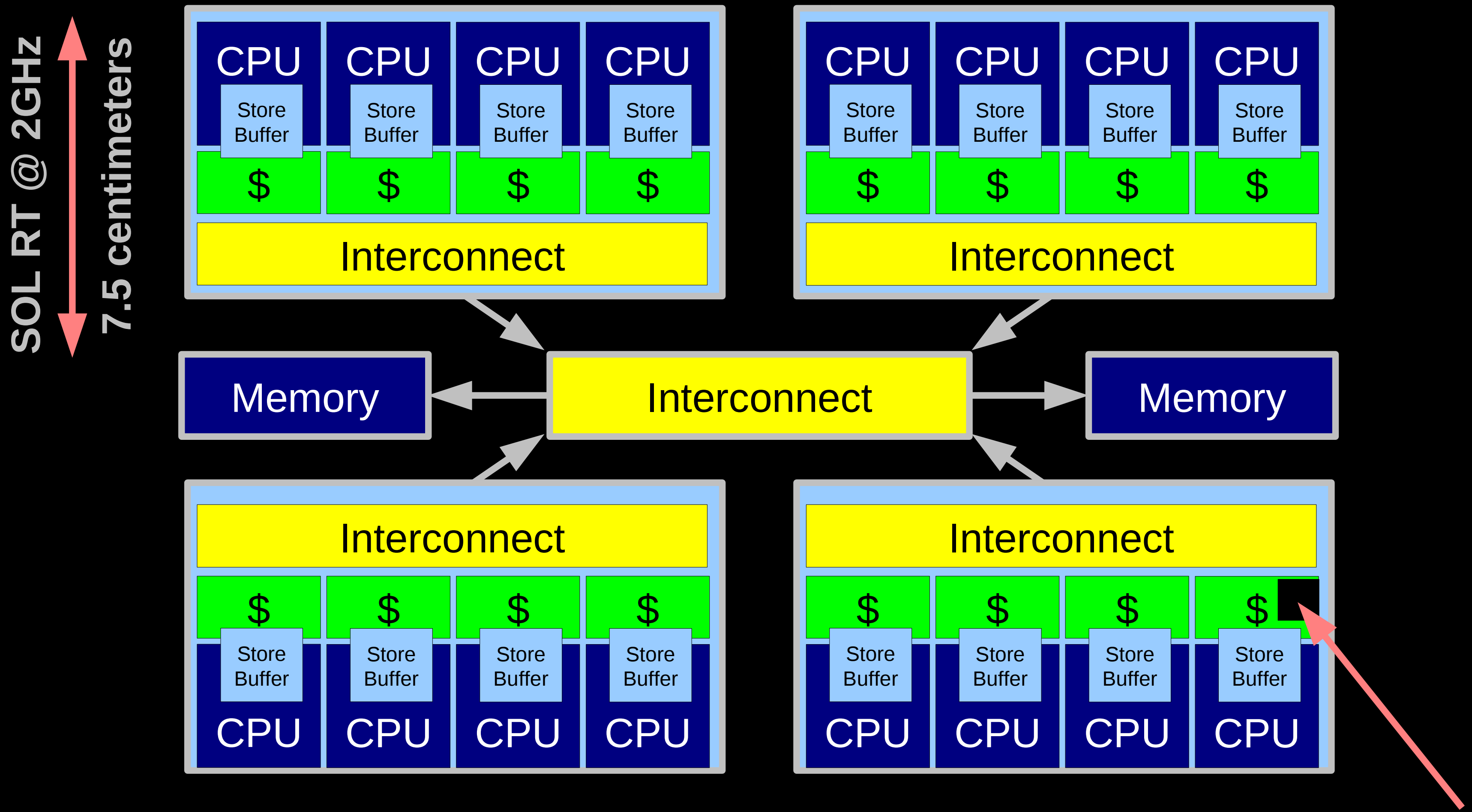
Observation by Stephen Hawking

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!

“Doctor, it Hurts When I Do Updates!!!

“Doctor, it Hurts When I Do Updates!!!

- “Then don't do updates!”

“Doctor, it Hurts When I Do Updates!!!

- “Then don't do updates!”
- “But if I don't do updates, I run out of registers!”

“Doctor, it Hurts When I Do Updates!!!

- “Then don't do updates!”
- “But if I don't do updates, I run out of registers!”

- We have no choice but to do updates, but we clearly need to be very careful with exactly ***how*** we do our updates

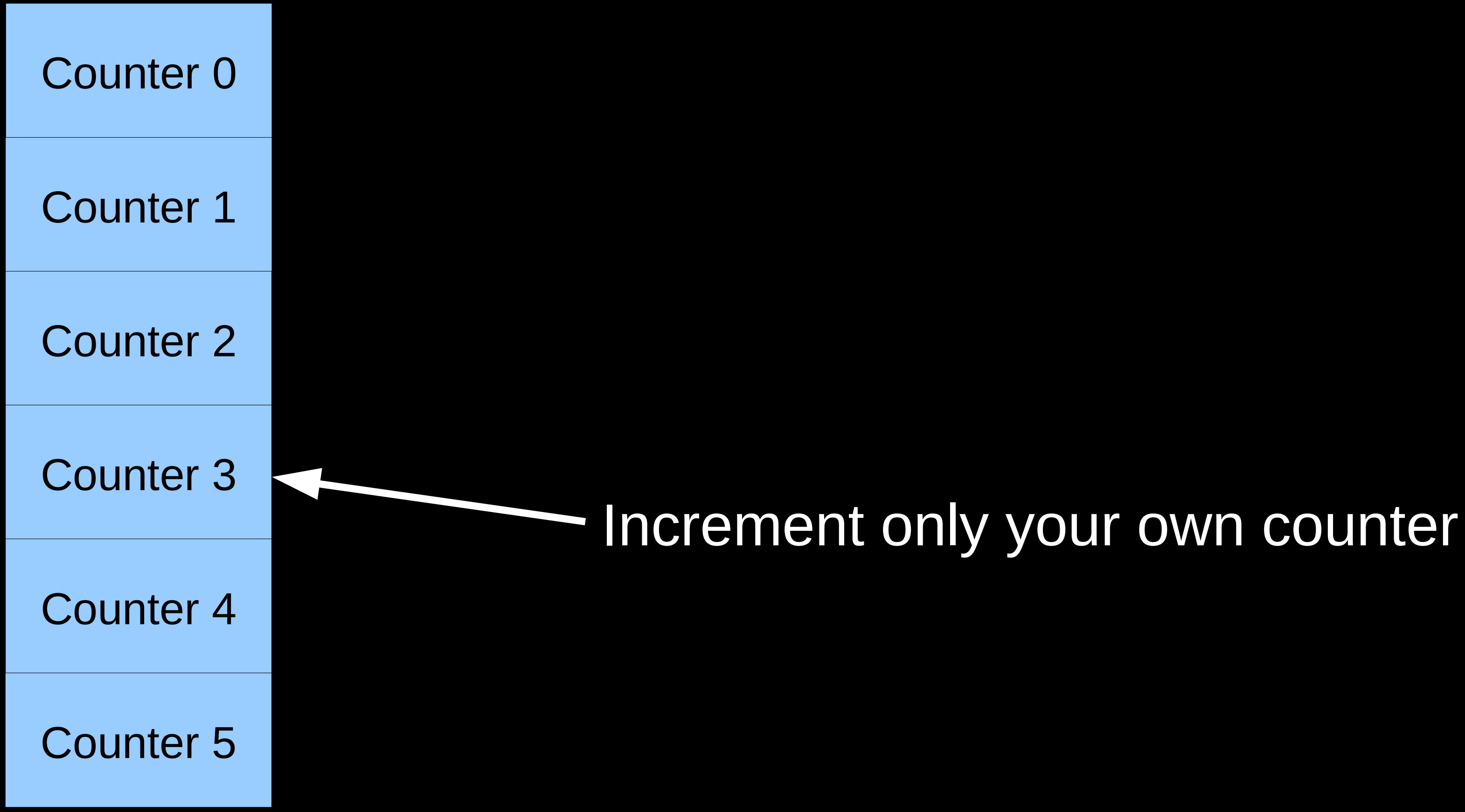
Update-Heavy Workloads Painful for Parallelism!!! But There Are Some Special Cases...

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



Counter 0

Counter 1

Counter 2

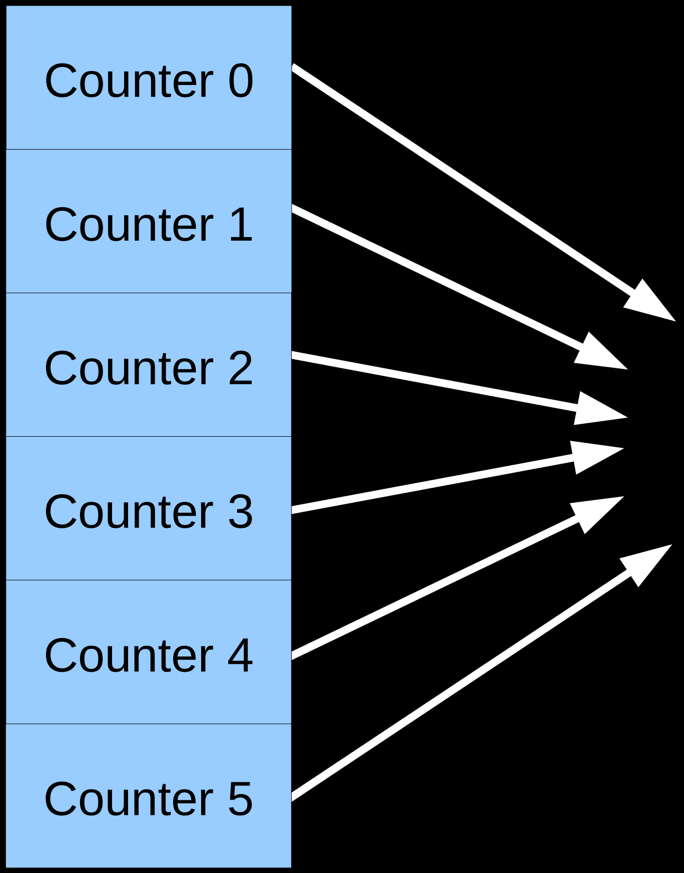
Counter 3

Counter 4

Counter 5

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)

Read-Only Traversal To Location Being Updated

Why Read-Only Traversal To Update Location?

- Consider a binary search tree
- Classic locking methodology would:
  1) Lock root
  2) Use key comparison to select descendant
  3) Lock descendant
  4) Unlock previous node
  5) Repeat from step (2)
- The lock contention on the root is not going to be pretty!
  - And we won't get contention-free moves of independent elements, so this cannot be a solution to the Issaquah Challenge

And This Is Why We Have RCU!

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

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

Quick overview, references at end of slideset.

And This Is Why We Have RCU!

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

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

- I assert that this gives the best possible performance, scalability, real-time response, wait-freedom, and energy efficiency

Quick overview, references at end of slideset.

And This Is Why We Have RCU!

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

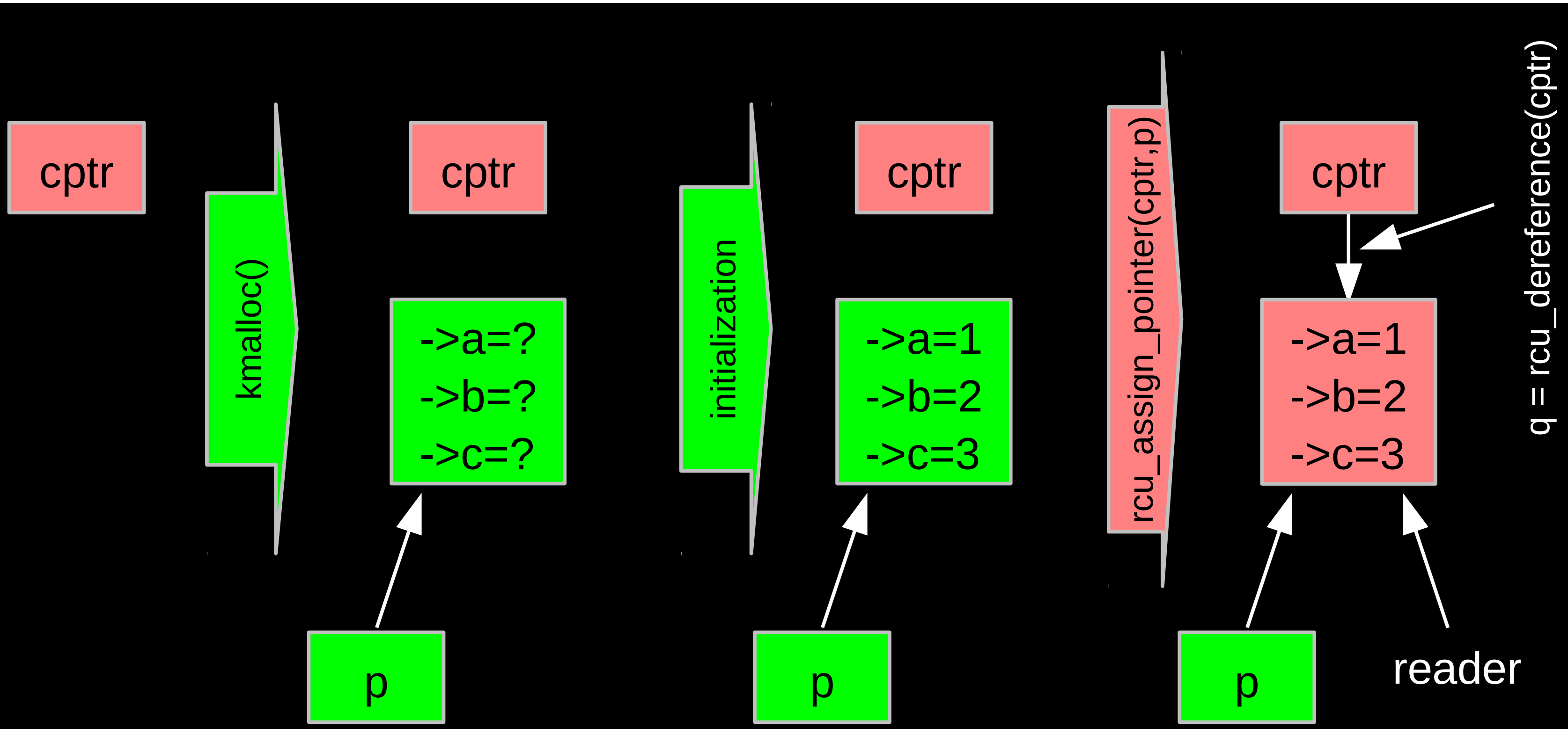
```
/* Assume non-preemptible (run-to-block) environment. */
#define rcu_read_lock()
#define rcu_read_unlock()
```

- I assert that this gives the best possible performance, scalability, real-time response, wait-freedom, and energy efficiency
- But 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

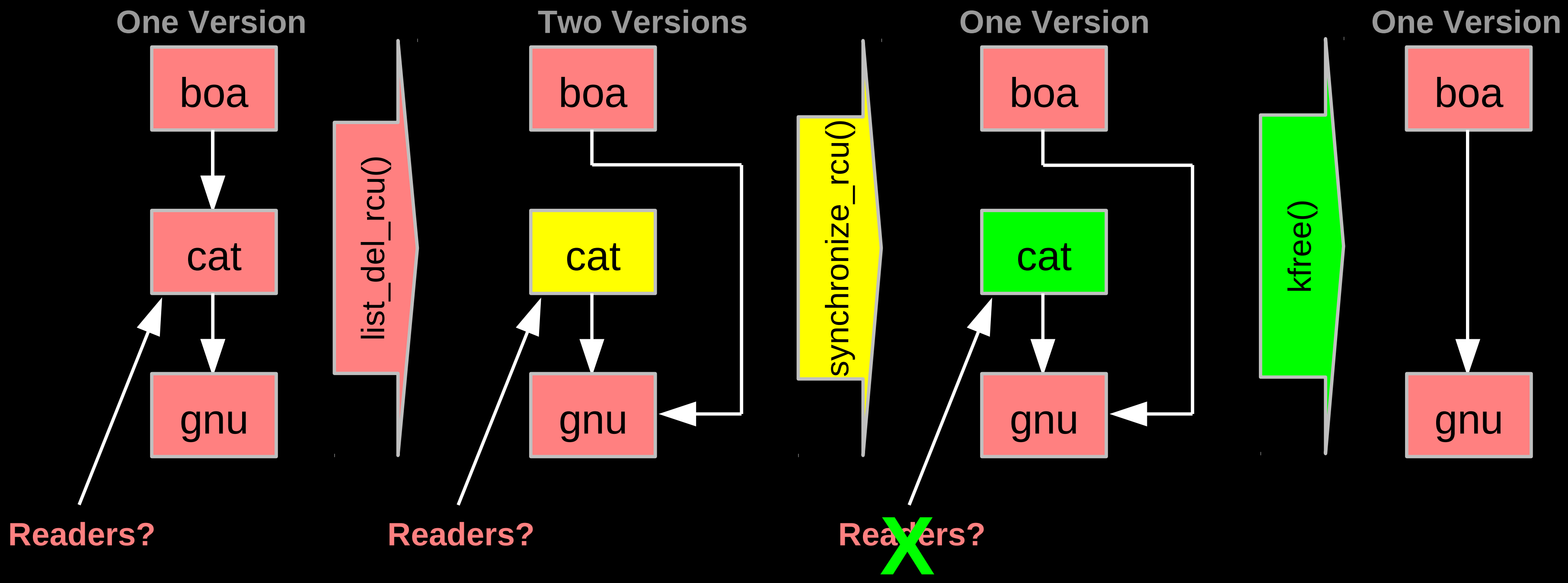
Key:
- 🟥 Dangerous for updates: all readers can access
- 🟨 Still dangerous for updates: pre-existing readers can access (next slide)
- 🟩 Safe for updates: inaccessible to all readers



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

RCU Safe Removal From Linked Structure

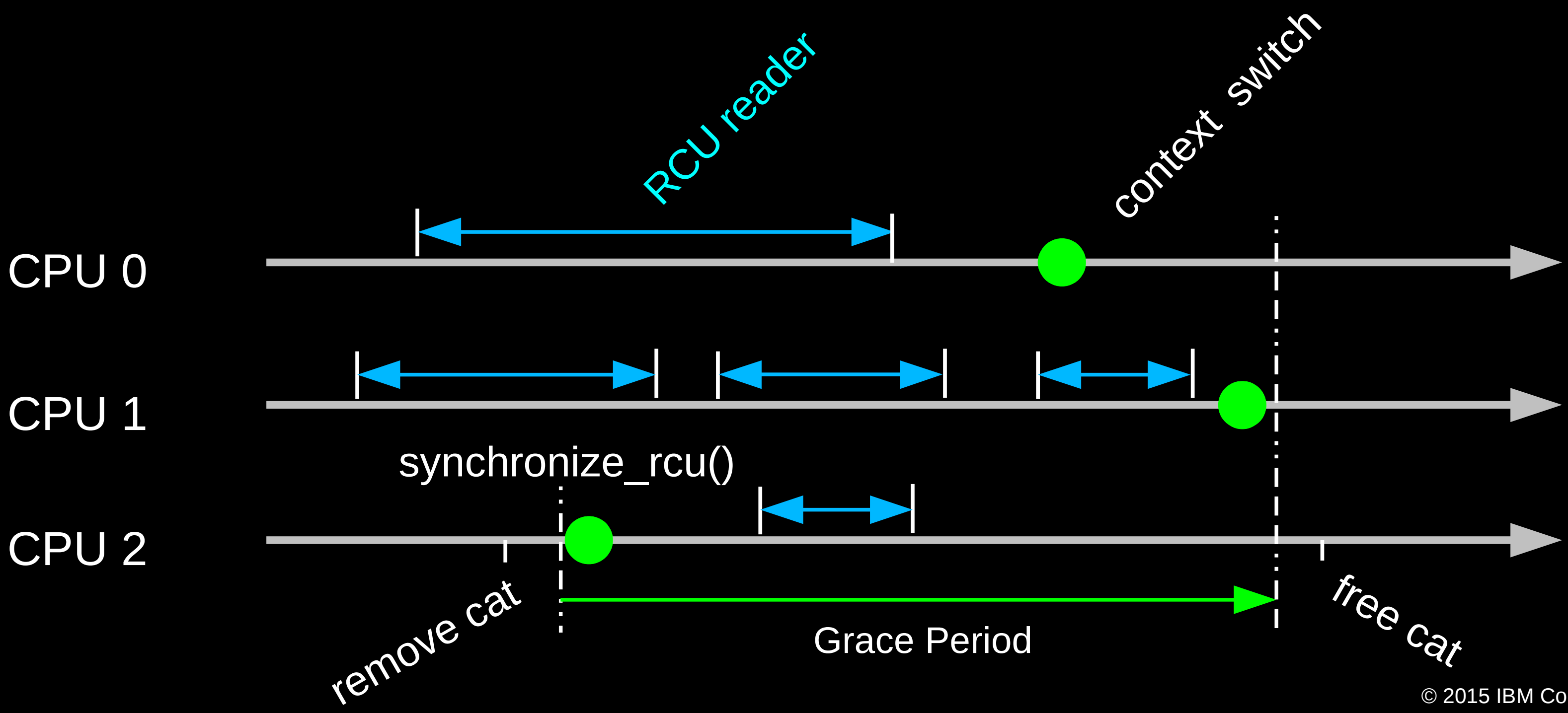
- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



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

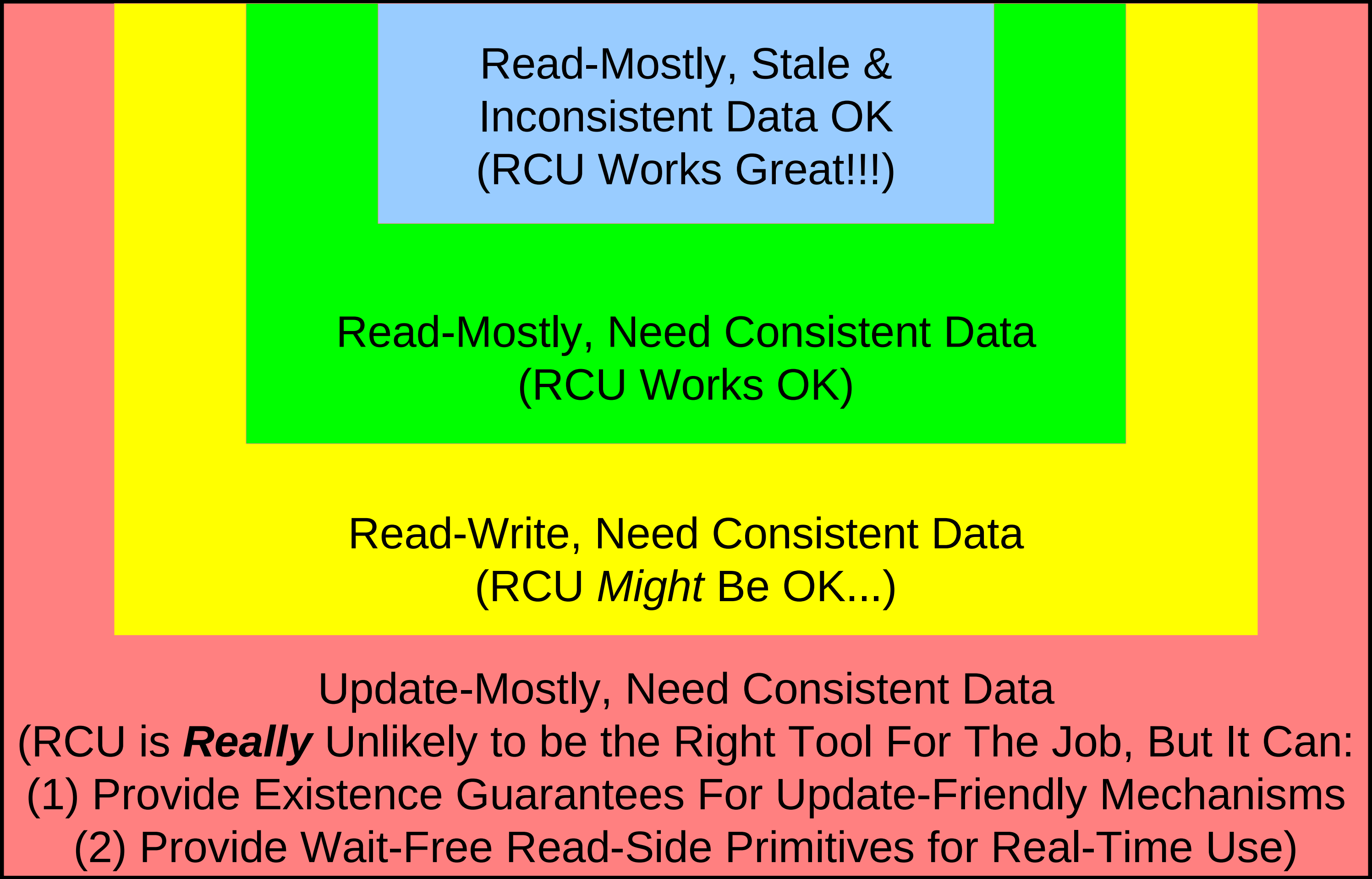
Synchronization Without Changing Machine State???

- 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”
- 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 Will Need Help...



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)

Better Read-Only Traversal To Update Location

Better Read-Only Traversal To Update Location

- An improved locking methodology might do the following:
  - rcu_read_lock()
  - Traversal:
    - Start at root without locking
    - Use key comparison to select descendant
    - Repeat until update location is reached
    - Acquire locks on update location
    - Do consistency checks, retry from root if inconsistent
  - Carry out update
  - rcu_read_unlock()
- Eliminates contention on root node!
- But need some sort of consistency-check mechanism...
  - RCU protects against freeing, not necessarily removal
  - “Removed” flags on individual data elements

Deletion-Flagged Read-Only Traversal

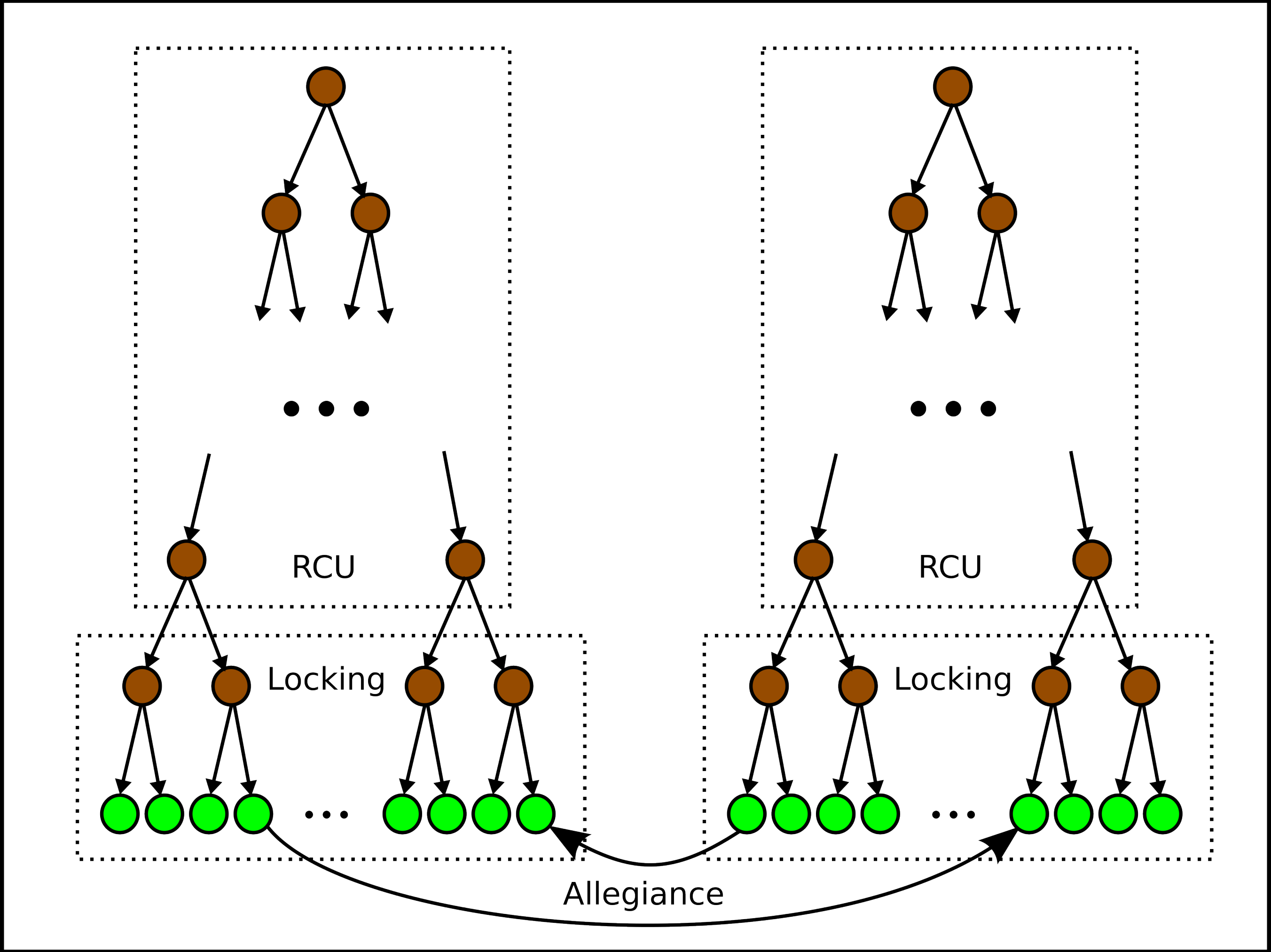
- for (;;)
  - rcu_read_lock()
  - Start at root without locking
  - Use key comparison to select descendant
  - Repeat until update location is reached
  - Acquire locks on update location
  - If to-be-updated location's “removed” flag is not set:
    - Break out of “for” loop
  - Release locks on update location
  - rcu_read_unlock()
- Carry out update
- Release locks on update location and rcu_read_unlock()

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

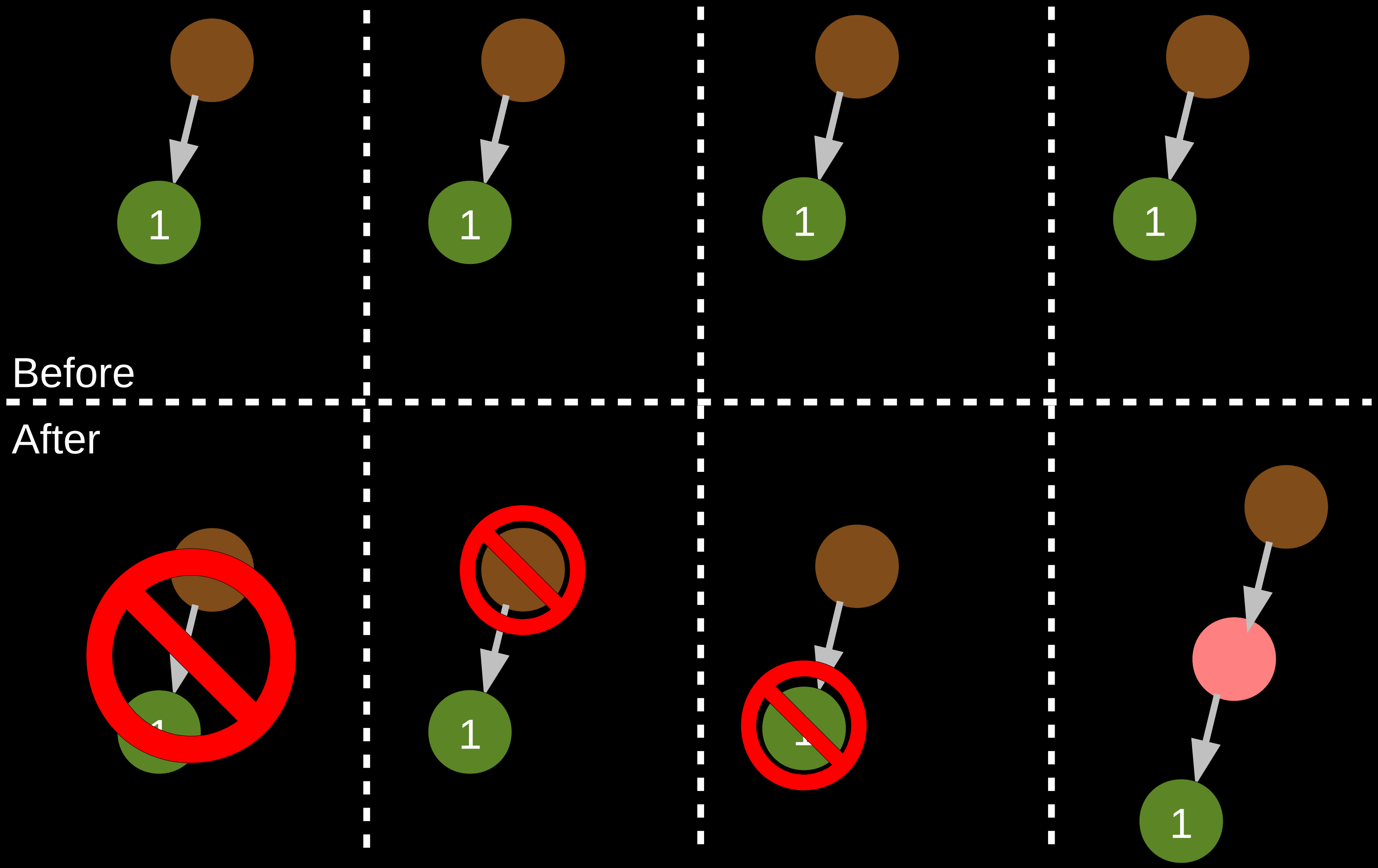
Issaquah Challenge: One Solution

Locking Regions for Binary Search Tree



Same tree algorithm with a few existence-oriented annotations

Possible Upsets While Acquiring Locks...



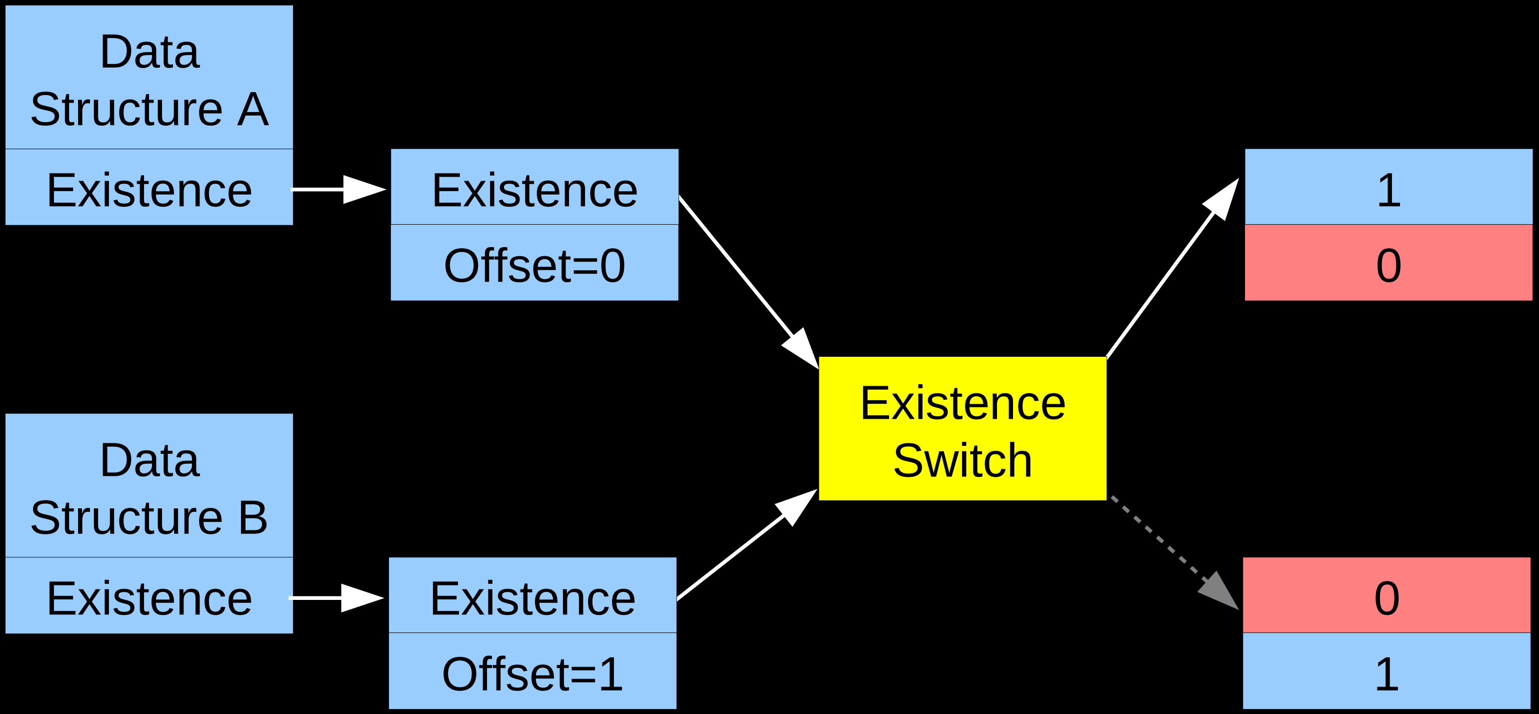
What to do?
Drop locks and retry!!!

Existence Structures

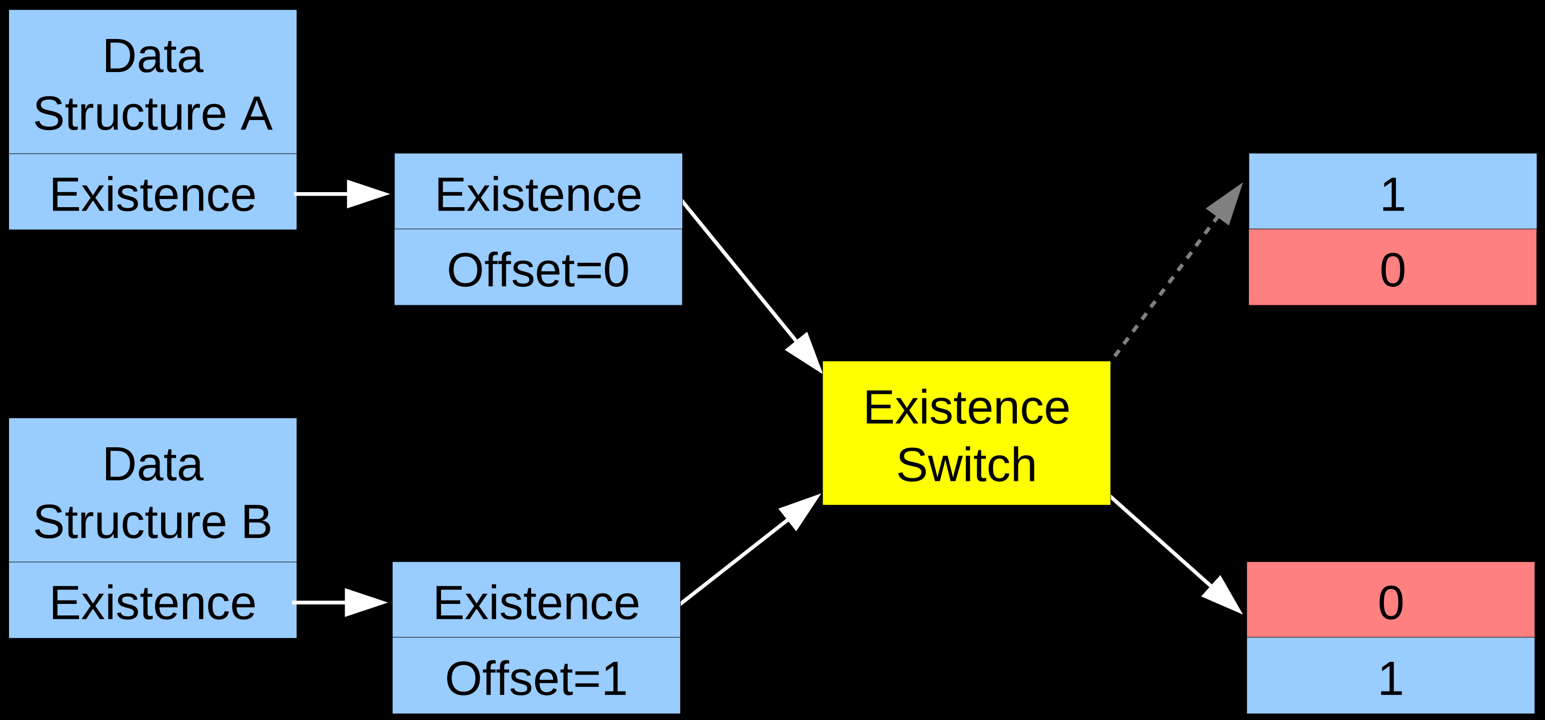
Existence Structures

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

Example Existence Structure Before Switch



Example Existence Structure After Switch



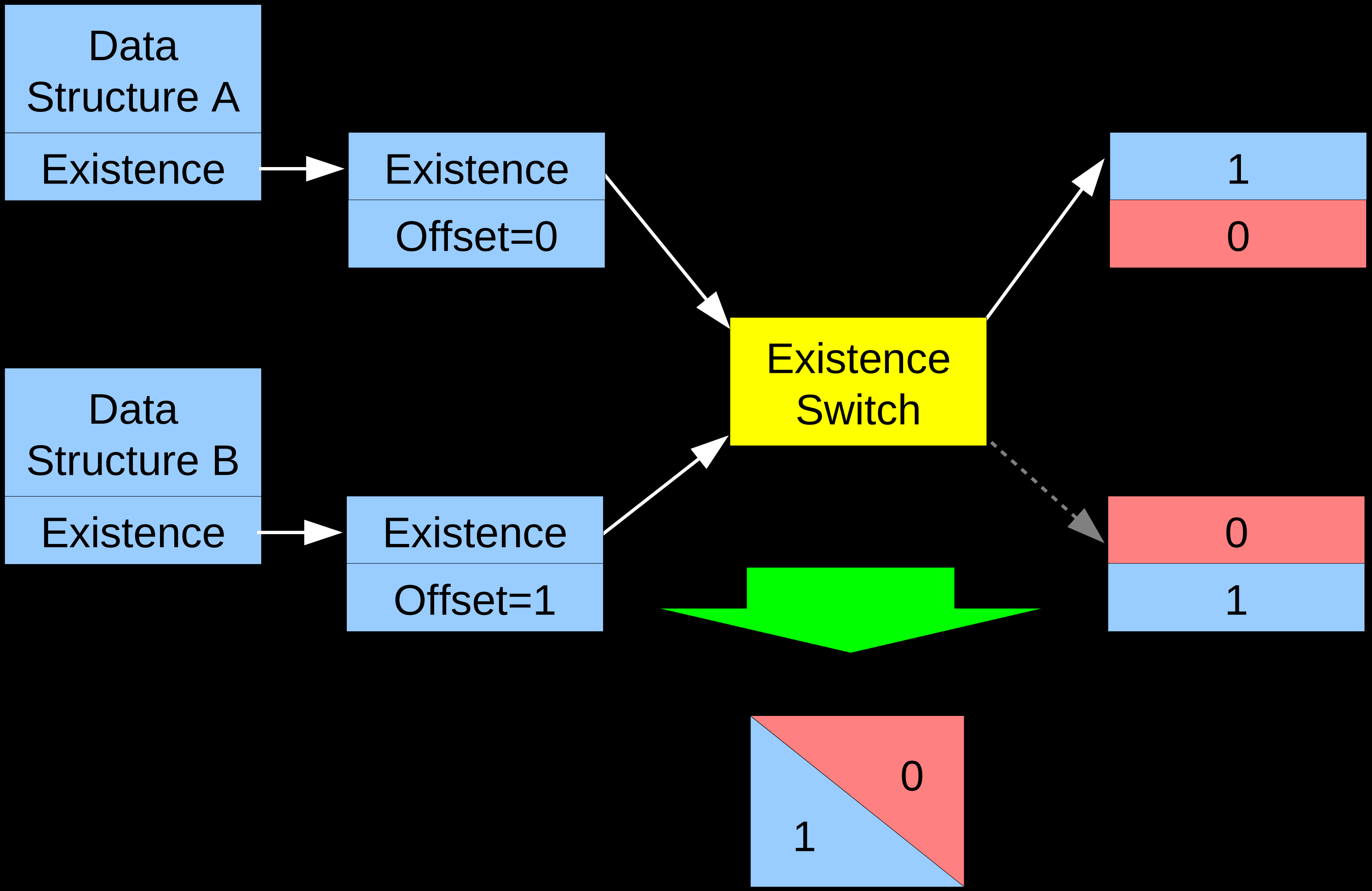
Existence Structure Definition

```
/* Existence-switch array. */
const int existence_array[4] = { 1, 0, 0, 1 };

/* Existence structure associated with each moving structure. */
struct existence {
        const int **existence_switch;
        int offset;
};

/* Existence-group structure associated with multi-structure change. */
struct existence_group {
        struct existence outgoing;
        struct existence incoming;
        const int *existence_switch;
        struct rcu_head rh;  /* Used by RCU asynchronous free. */
};
```

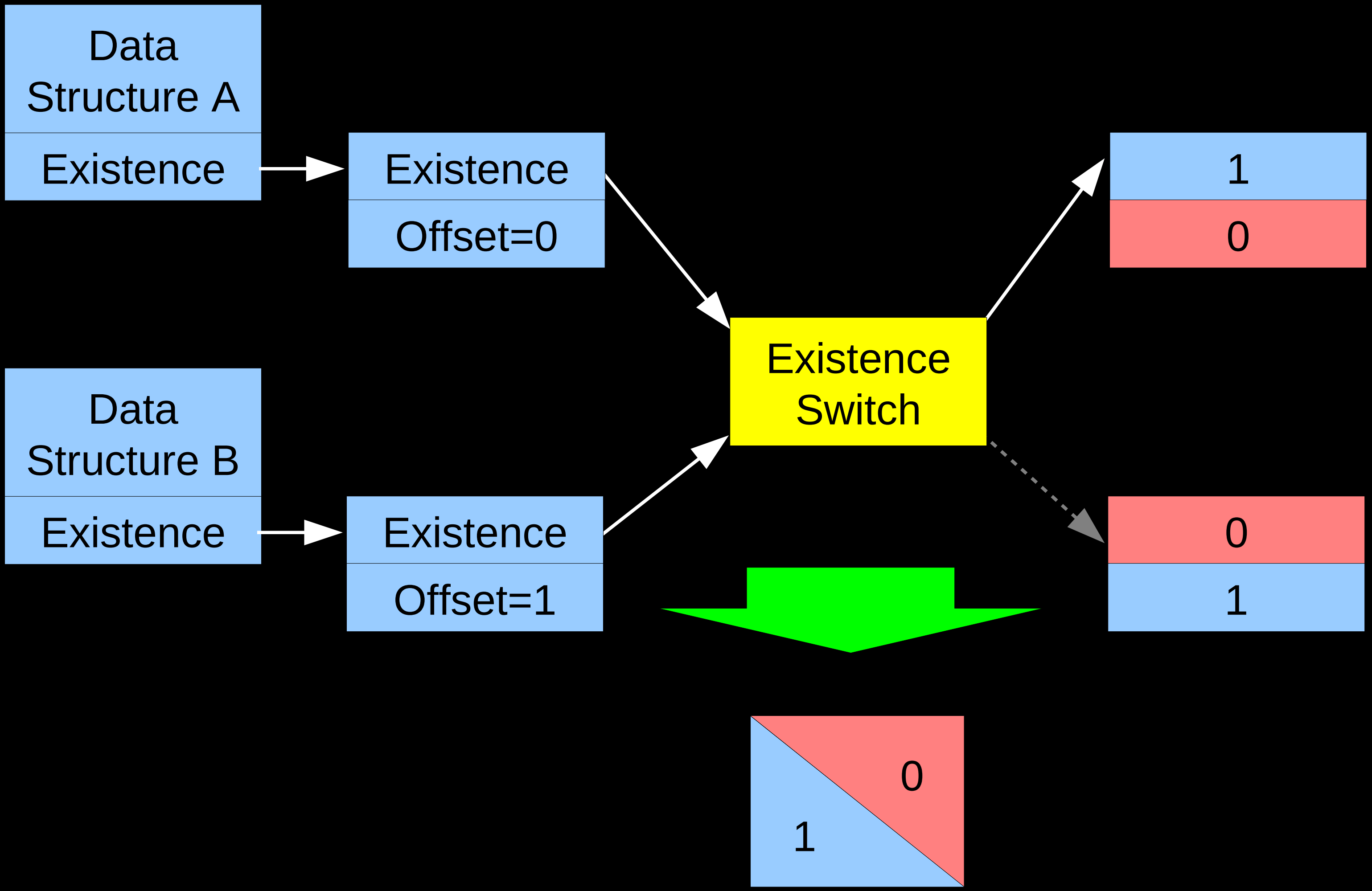
Example Existence Structure: Abbreviation



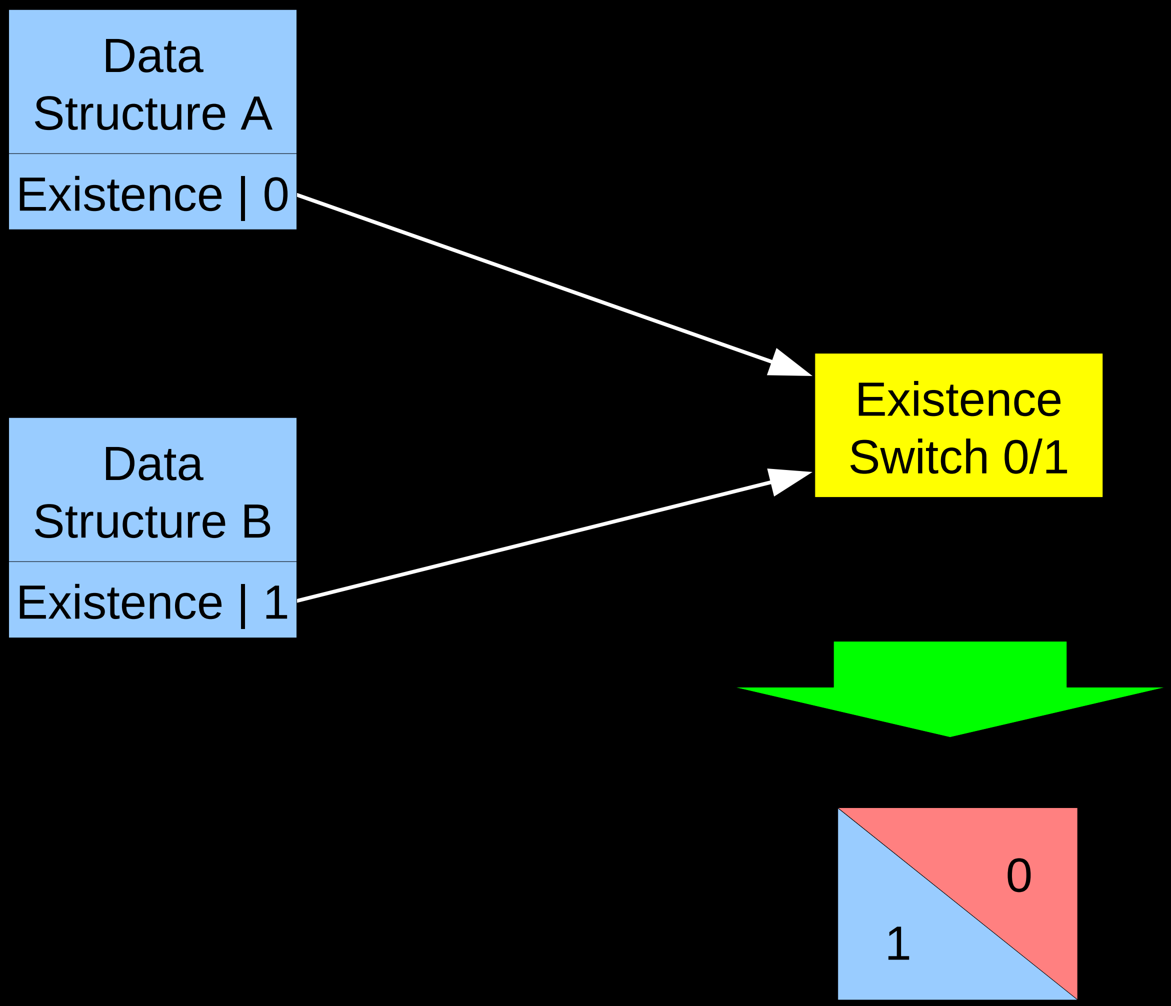
But Levels of Indirection Are Expensive!

- 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, see next slide

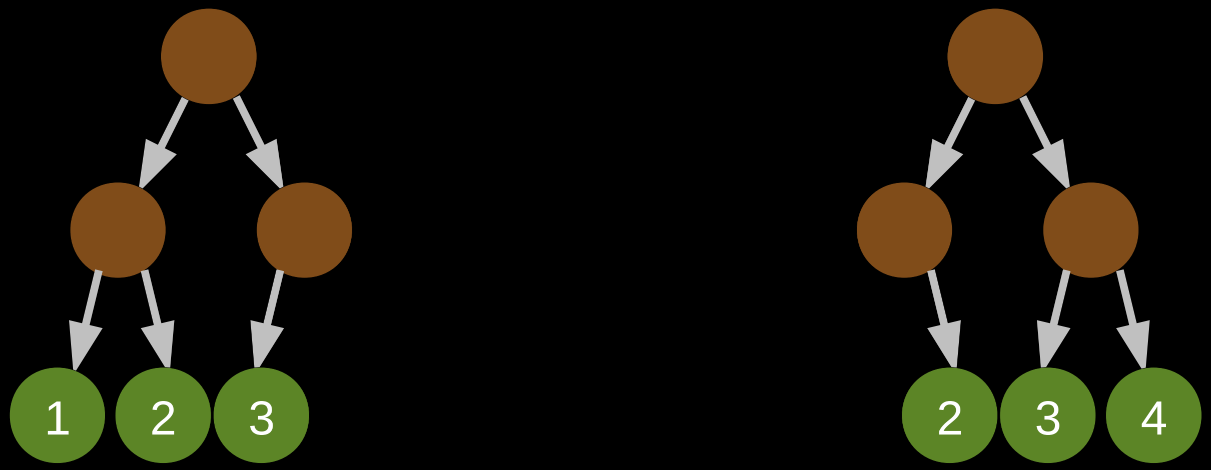
Example Existence Structure: Before Dmitry



Example Existence Structure: After Dmitry

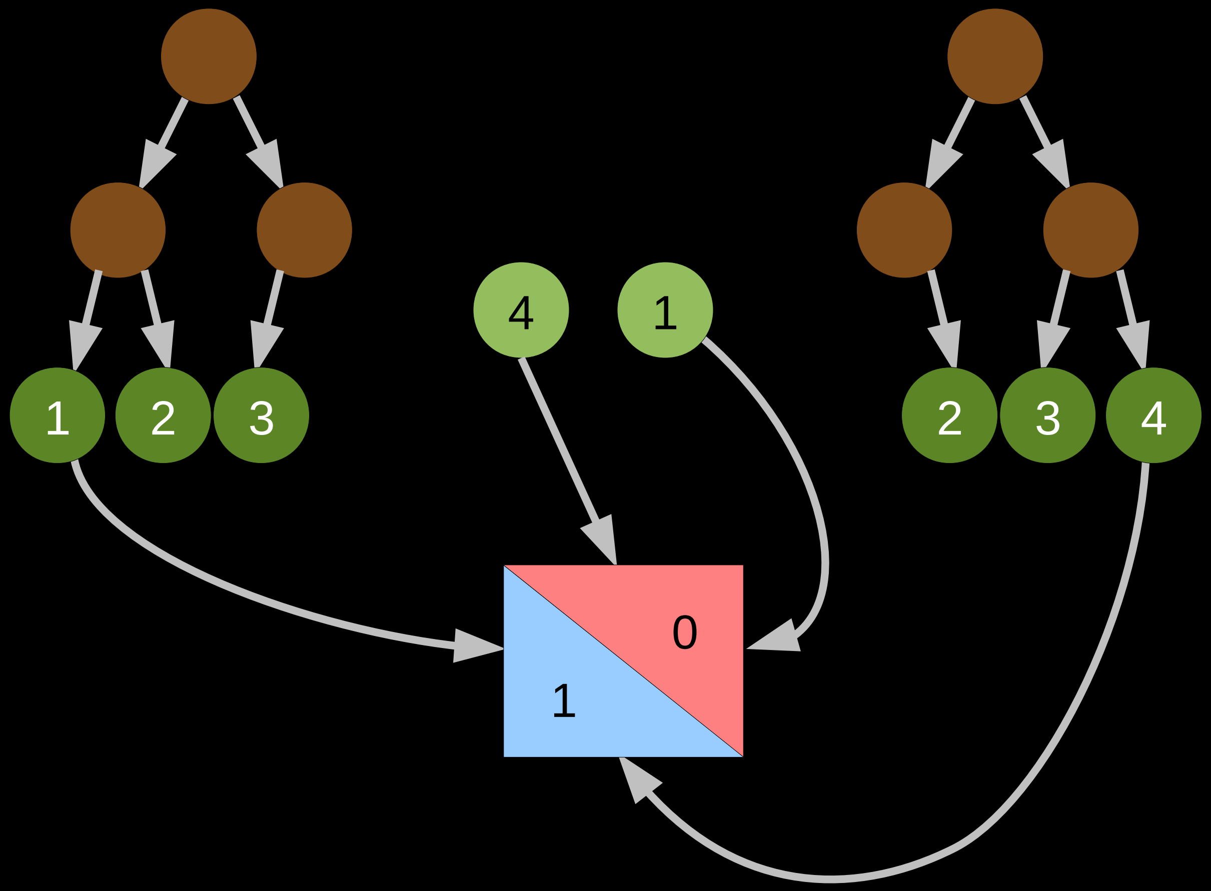


Abbreviated Existence Switch Operation (1/6)



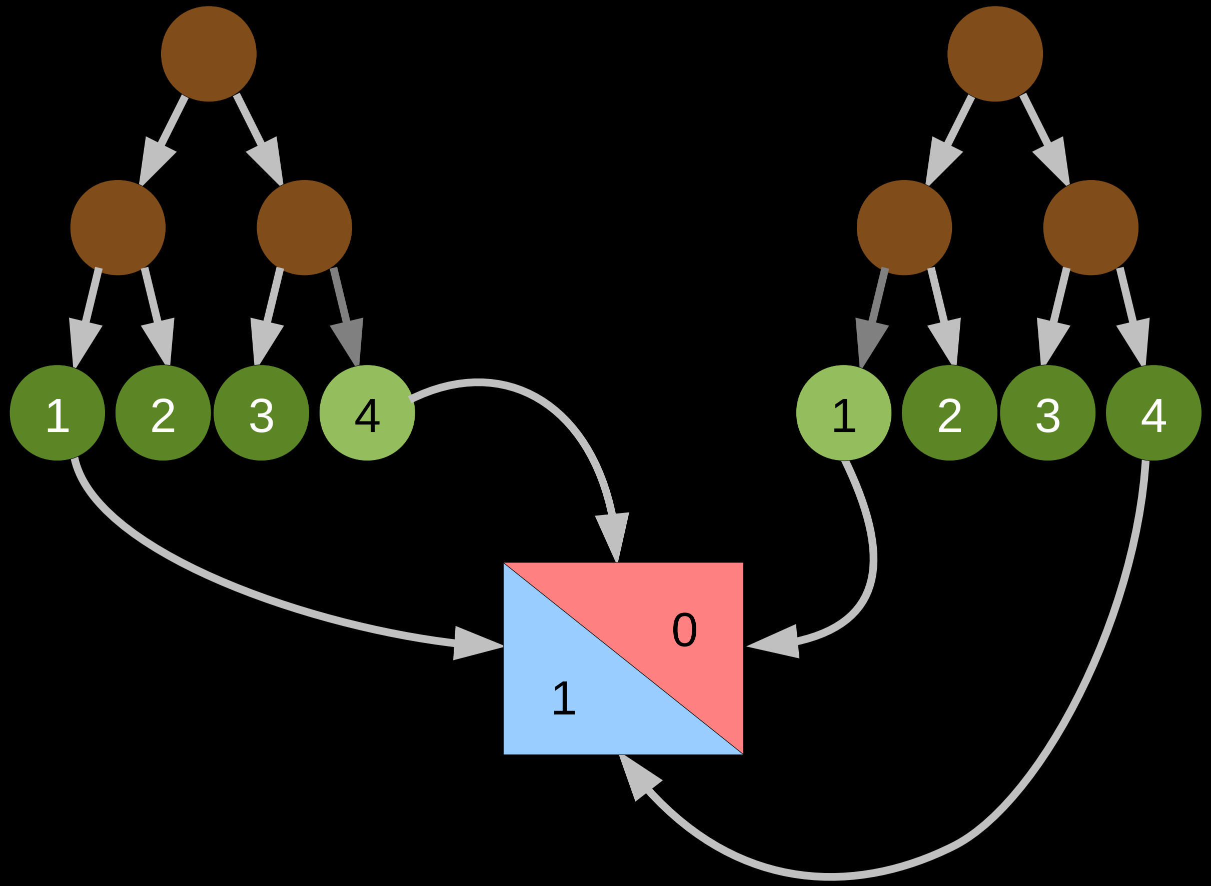
Initial state: First tree contains 1,2,3, second tree contains 2,3,4.
All existence pointers are NULL.

Abbreviated Existence Switch Operation (2/6)



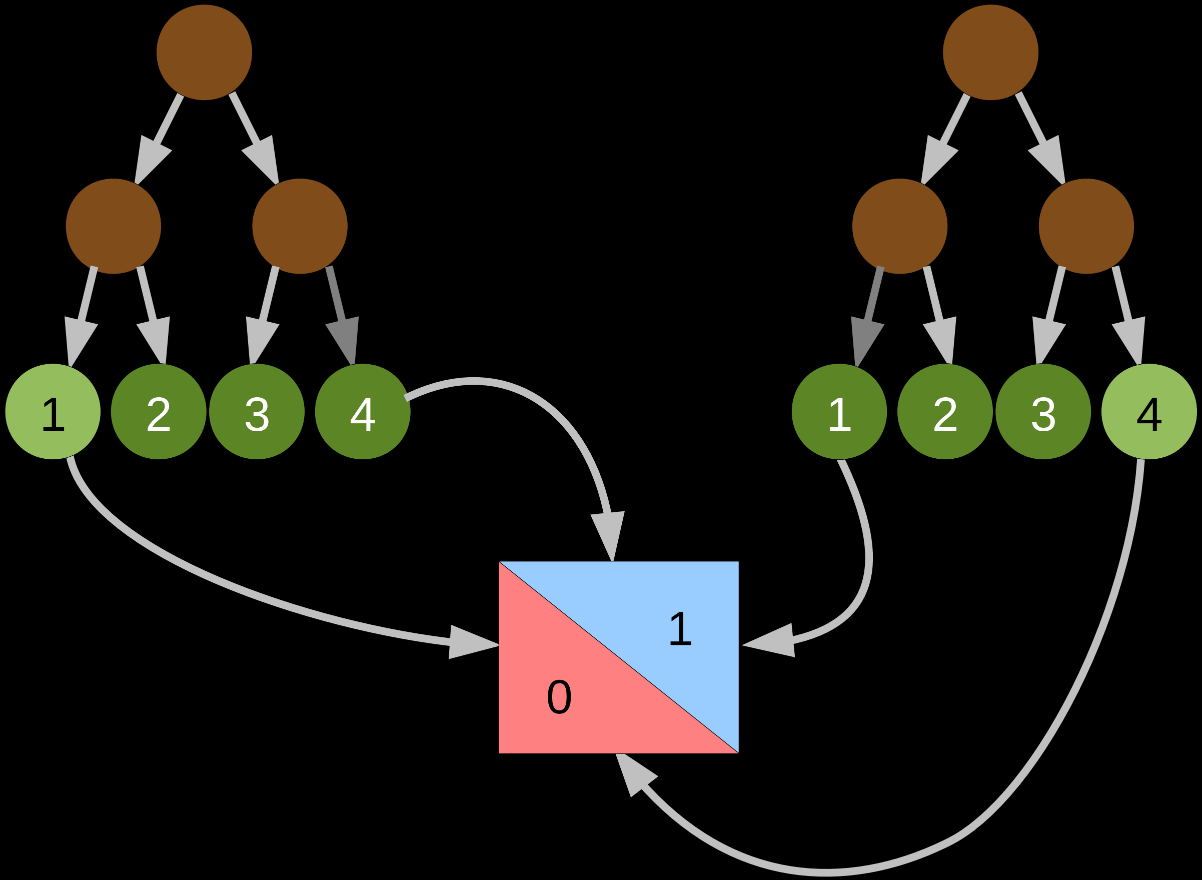
First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (3/6)



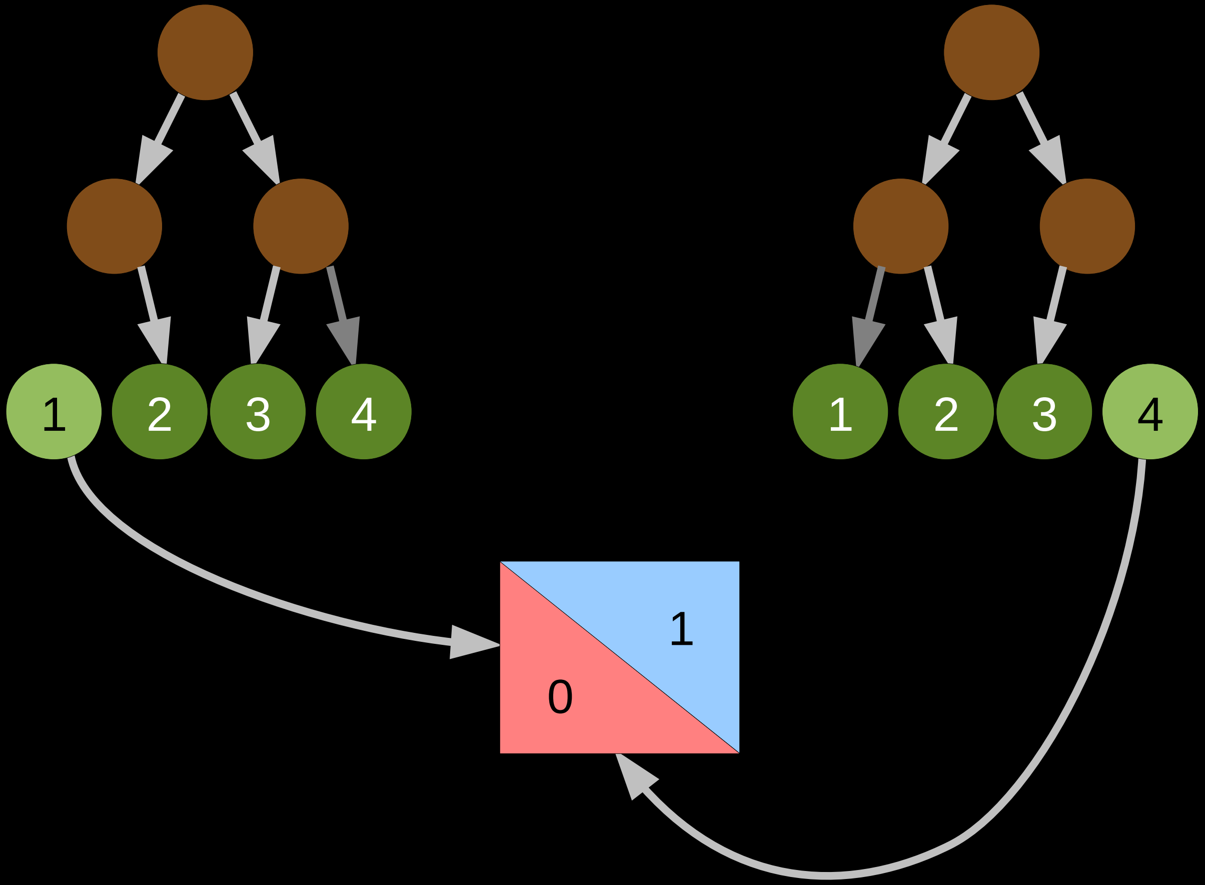
After insertion, same: First tree contains 1,2,3, second tree contains 2,3,4.

Abbreviated Existence Switch Operation (4/6)



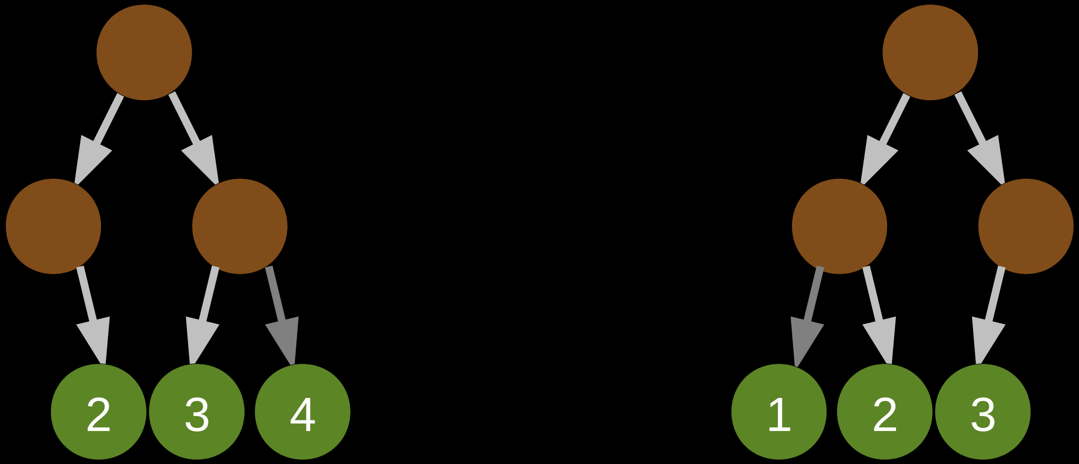
After existence switch: First tree contains 2,3,4, second tree contains 1,2,3.
Transition is single store, thus atomic! (But lookups need barriers in this case.)

Abbreviated Existence Switch Operation (5/6)



Unlink old nodes and allegiance structure

Abbreviated 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
    - 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 APIs

- `struct existence_group *existence_alloc(void);`
- `void existence_free(struct existence_group *egp);`
- `struct existence *existence_get_outgoing(struct existence_group *egp);`
- `struct existence *existence_get_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 **epp);`
- `int existence_exists_relaxed(struct existence **epp);`
- `void existence_switch(struct existence_group *egp);`

Existence Operations for Trees

-
  ```
  int tree_atomic_move(struct treeroot *srcp, struct treeroot *dstp,
                       int key, void **data_in)
  ```

-
  ```
  int tree_existence_add(struct treeroot *trp, int key,
                                struct existence *ep, void **data)
  ```

-
  ```
  int tree_existence_remove(struct treeroot *trp, int key,
                                   struct existence *ep)
  ```

-
  ```
  int tree_insert_existence(struct treeroot *trp, int key, void *data,
                            struct existence *node_existence, int wait)
  ```

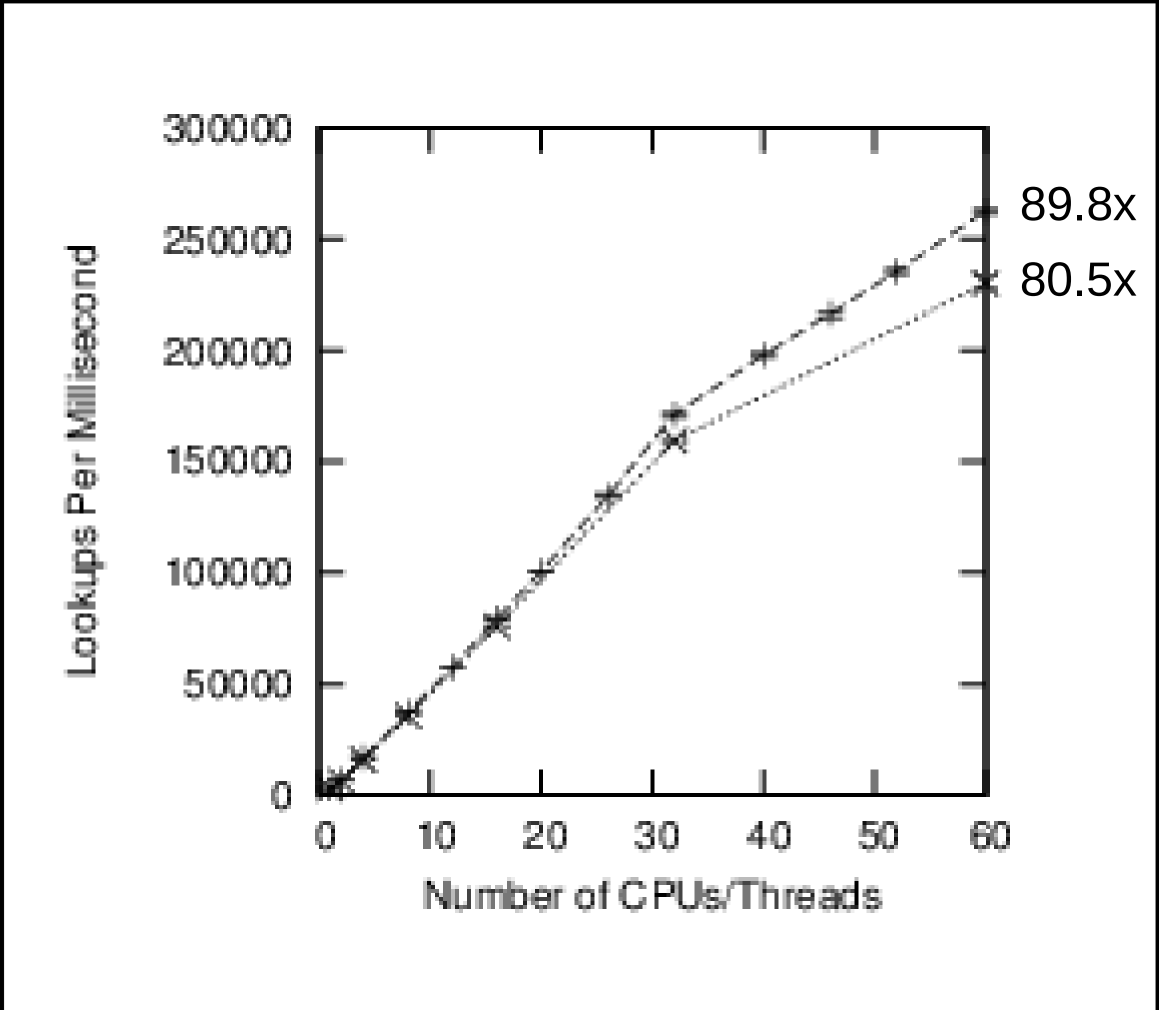
-
  ```
  int tree_delete_existence(struct treeroot *trp, int key,
                                   void **data, void *matchexistence, int wait)
  ```

Same tree algorithm with a few existence-oriented annotations

Pseudo-Code for Atomic Tree Move

- Allocate existence_group structure (existence_alloc())
- Add outgoing existence structure to item in source tree (existence_set())
  - If operation fails, report error to caller
- Insert new element (with source item's data pointer) to destination tree with incoming existence structure (variant of tree_insert())
  - If operation fails, remove existence structure from item in source tree, free and report error to caller
- Invoke existence_switch() to flip incoming and outgoing
- Delete item from source tree (variant of tree_delete())
- Remove existence structure from item in destination tree (existence_clear())
- Free existence_group structure (existence_free())

Existence Structures: Performance and Scalability



89.8x LCA

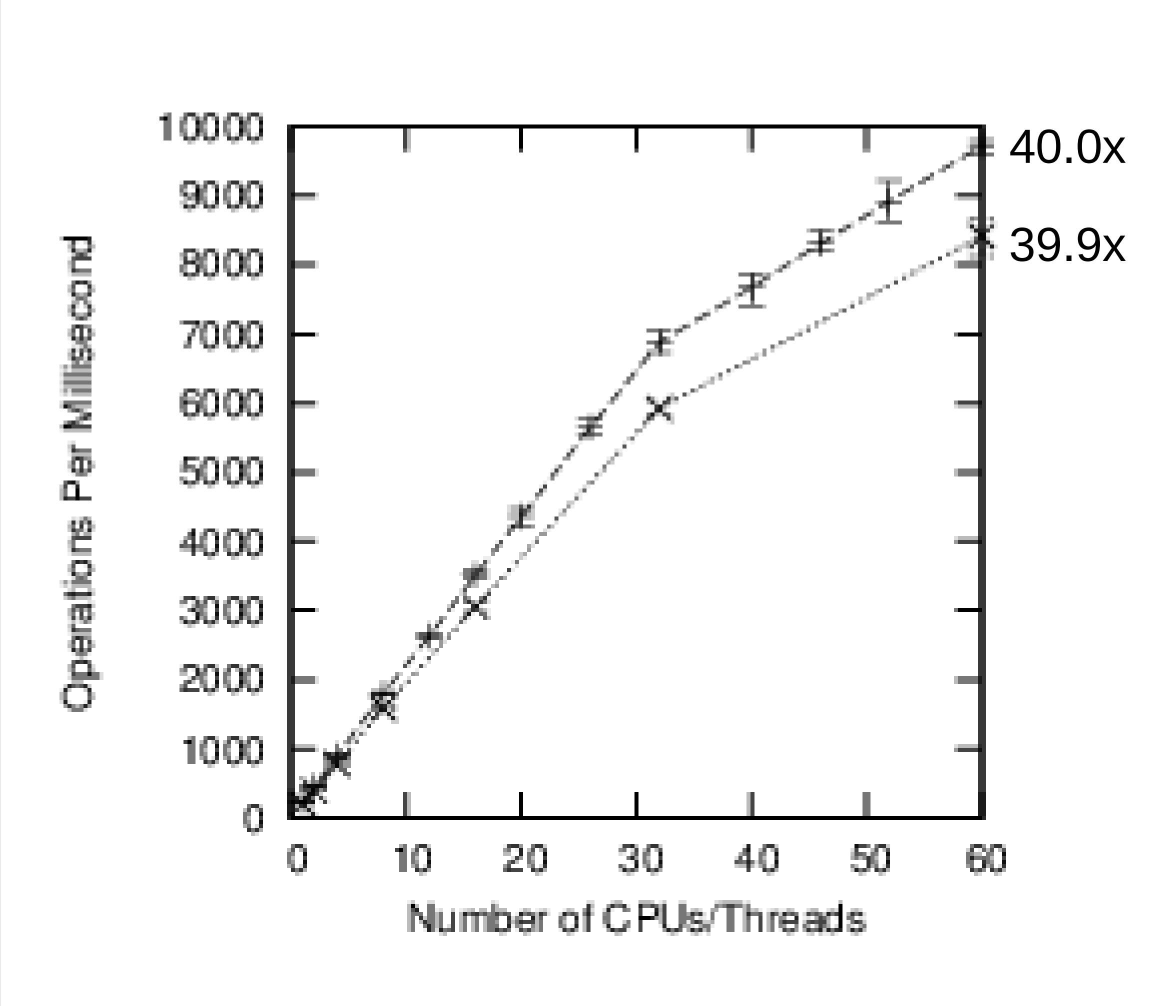
80.5x CPPCON

100% lookups

Super-linear as expected based on range partitioning

(Hash tables about 3x faster)

Existence Structures: Performance and Scalability

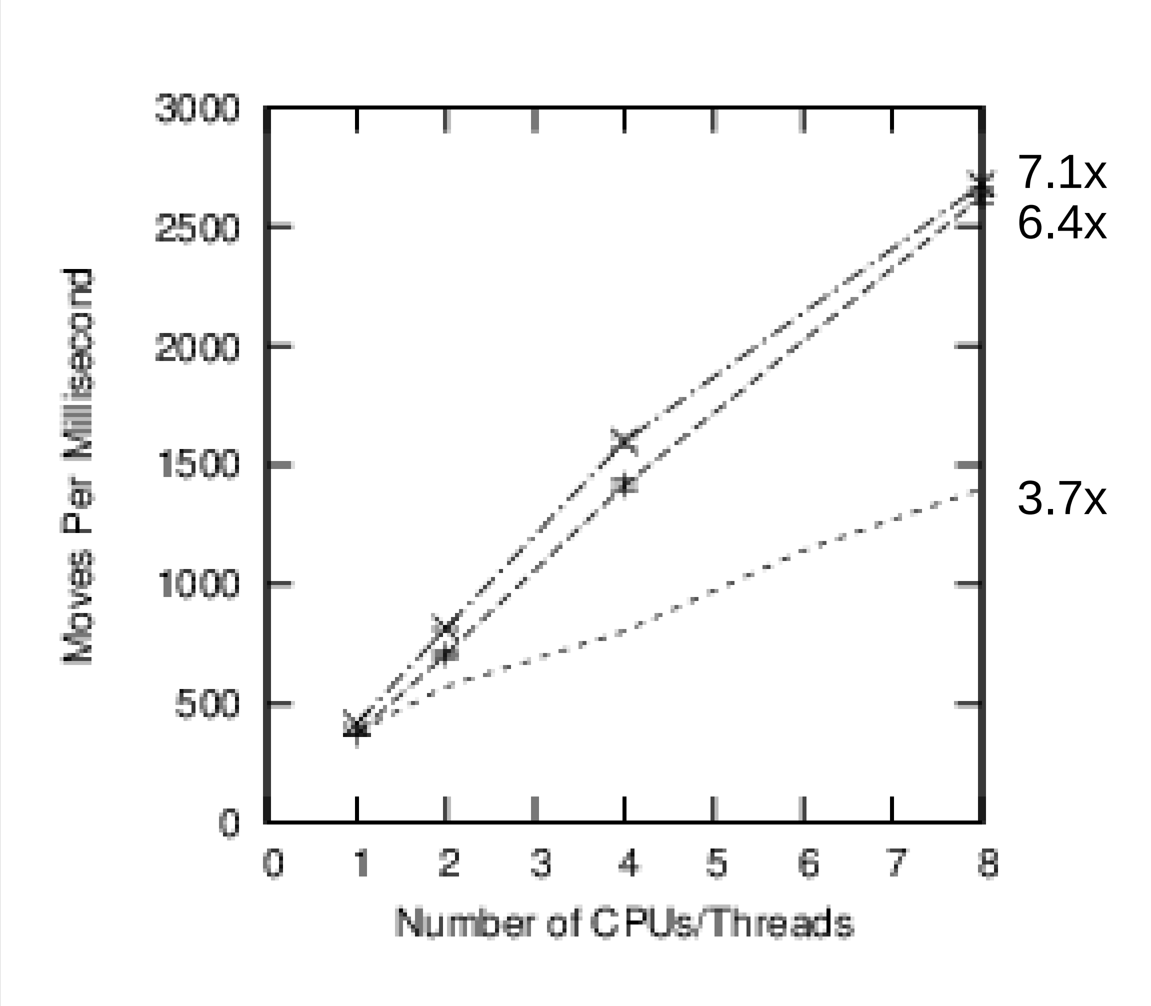


40.0x LCA

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



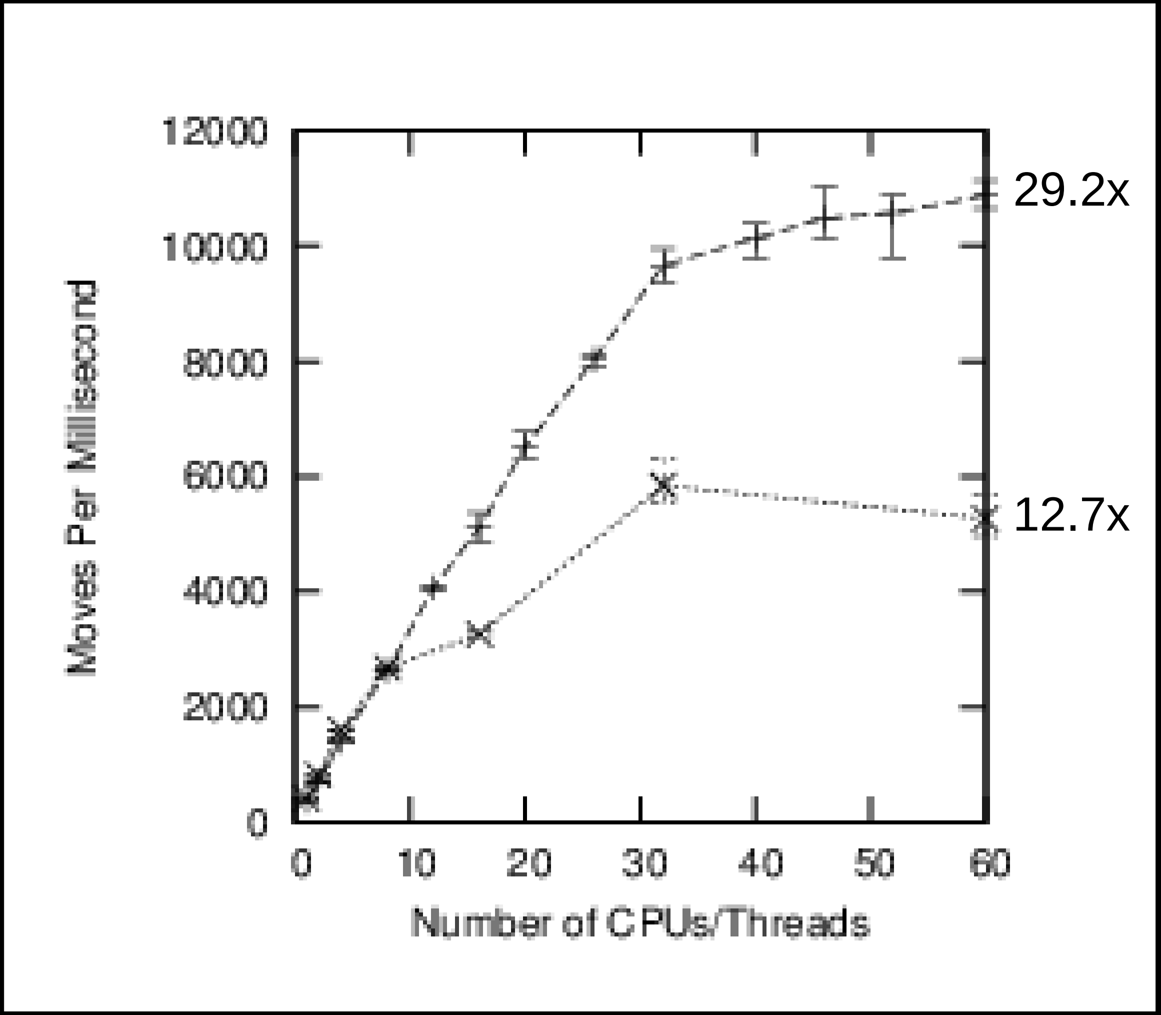
LCA

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...

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, with apologies to Kathleen Turner fans
- Issues thus far:
  - Getting possible-upset checks right
  - Non-scalable random-number generator
  - Non-scalable memory allocator
  - Node alignment (false sharing)
  - Premature deletion of moved elements (need to remove allegiance!)
  - Unbalanced trees (false sharing)
  - User-space RCU configuration (need per-thread call_rcu() handling)
  - Getting memory barriers correct (probably more needed here)
  - Threads working concurrently on adjacent elements (false sharing)
  - Need to preload destination tree for move operations (contention!)
  - Issues from less-scalable old version of user-space RCU library
  - More memory-allocation tuning
  - Wakeup interface to user-space RCU library (instead of polling)
  - More URCU tuning
- Next steps: More detailed profiling for poorly scaling scenarios

Existence Advantages and Disadvantages

- Existence requires focused developer effort
- Existence specialized to linked structures (for now, anyway)
- Existence requires explicit memory management
  - Might eventually be compatible with shared pointer, but not yet
- Existence-based exchange operations require linked structures that accommodate duplicate elements
  - Current prototypes disallow duplicates
- Existence permits irrevocable operations
- Existence can exploit locking hierarchies, reducing the need for contention management
- Existence achieves semi-decent performance and scalability
- 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!!!

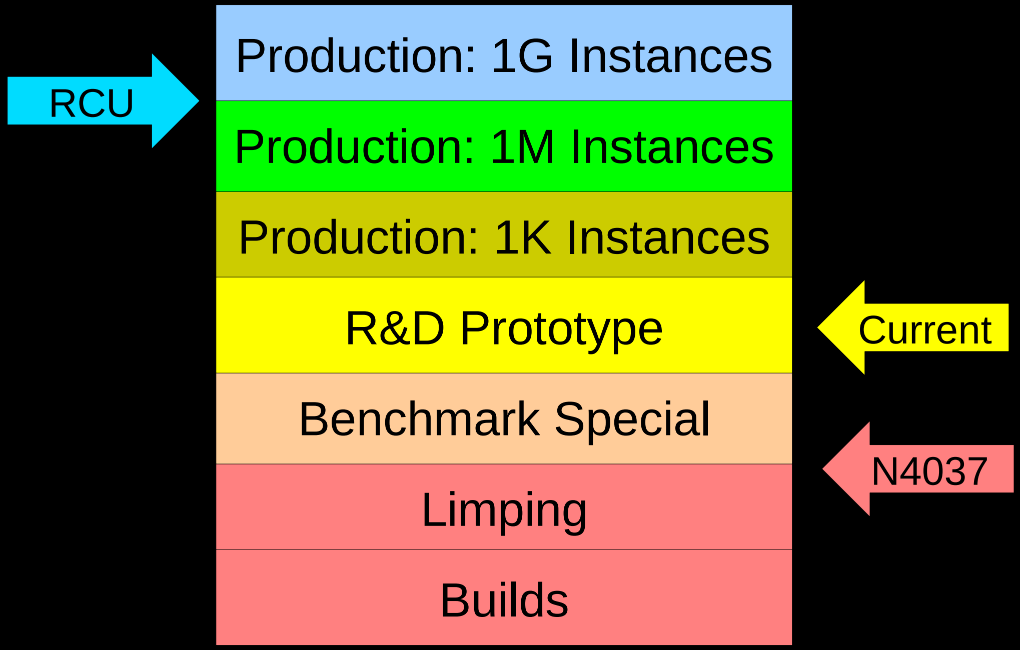
When Might You Use Existence-Based Update?

- 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
  - Need compatibility with hardware not supporting transactional memory
  - Need to be able to do irrevocable operations (e.g., I/O) as part of data-structure update

Existence Structures: Production Readiness

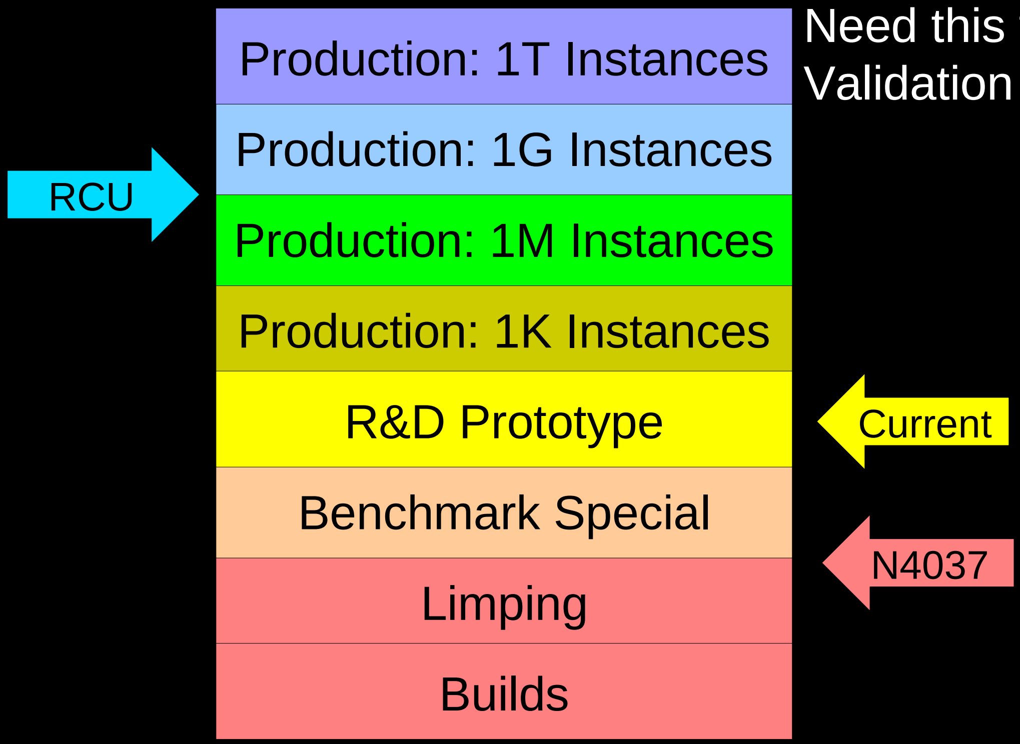
Existence Structures: Production Readiness

- No, it is ***not*** production ready (but getting there)
  - In happy contrast to a few months ago...



Existence Structures: Production Readiness

- No, it is ***not*** production ready (but getting there)
  - In happy contrast to a few months ago...



Production: 1T Instances

Production: 1G Instances

Production: 1M Instances

Production: 1K Instances

R&D Prototype

Benchmark Special

Limping

Builds

RCU

Current

N4037

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

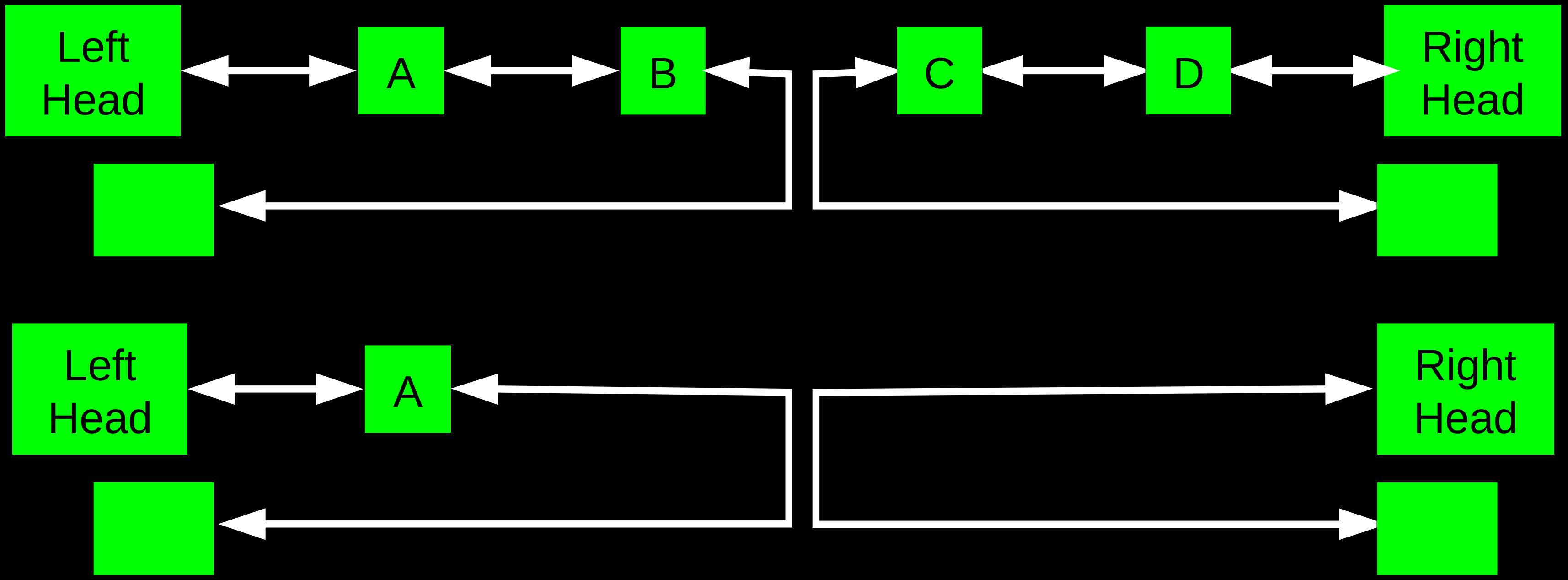
Existence Structures: Known Antecedents

- 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

Trivial Lock-Based Concurrent Deque

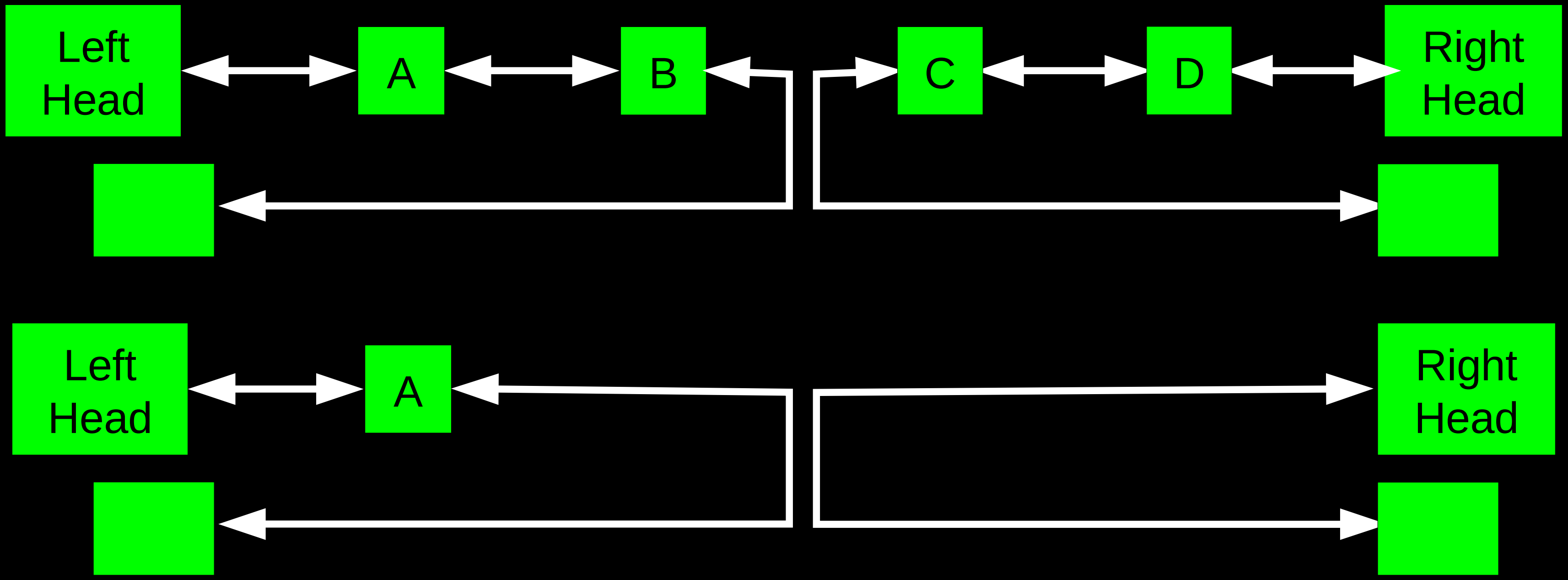
Trivial Lock-Based Concurrent Deque

- 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 Deque

- 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 deque???



Summary

Summary

- There is currently no silver bullet:
  - Split counters
    - Extremely specialized
  - Per-CPU/thread processing
    - Not all algorithms can be efficiently partitioned
  - Stream-based applications
    - Specialized
  - Read-only traversal to location being updated
    - Great for small updates to large data structures, but limited otherwise
  - Hardware lock elision
    - Some good potential, and some potential limitations
- Linux kernel: Good progress by combining approaches
- 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?

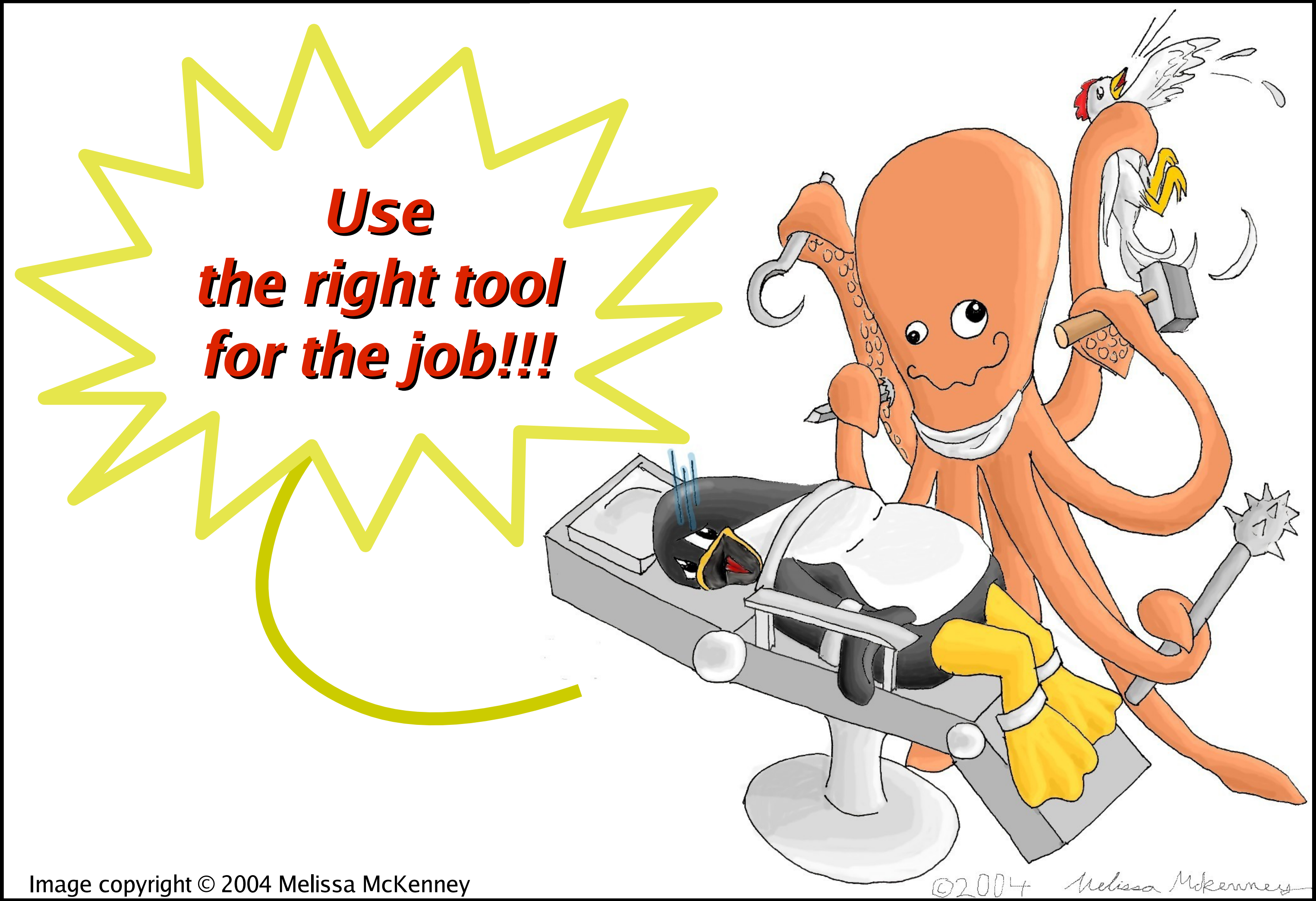


Image copyright © 2004 Melissa McKenney

BACKUP

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

- Pointers to RCU-protected objects are guaranteed to exist throughout a given RCU read-side critical section

```
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr); /* consume load */
/* *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 use more 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; /* Can be relaxed load. */
rcu_assign_pointer(cptr, newp); /* store release */
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

Hardware Lock Elision: Potential Game Changers

What must happen for HTM to take over the world?

Hardware Lock Elision: Potential Game Changers

- Forward-progress guarantees
  - Mainframe is a start, but larger sizes would be helpful
- Transaction-size increases
- Improved debugging support
  - Gottschich et al: “But how do we really debug transactional memory?”
- Handle irrevocable operations (unbuffered I/O, syscalls, ...)
- Weak atomicity

Hardware Lock Elision: Potential Game Changers

- Forward-progress guarantees
  - Mainframe is a start, but larger sizes would be helpful
- Transaction-size increases
- Improved debugging support
  - Gottschich et al: “But how do we really debug transactional memory?”
- Handle irrevocable operations (unbuffered I/O, syscalls, ...)
- Weak atomicity – but of course the Linux-kernel RCU maintainer and weak-memory advocate ***would*** say that...

Hardware Lock Elision: Potential Game Changers

- Forward-progress guarantees
  - Mainframe is a start, but larger sizes would be helpful
- Transaction-size increases
- Improved debugging support
  - Gottschich et al: “But how do we really debug transactional memory?”
- Handle irrevocable operations (unbuffered I/O, syscalls, ...)
- Weak atomicity: It is not just me saying this!
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
  - Shavit: “Data structures in the multicore age”
  - Haas et al: “How FIFO is your FIFO queue?”
  - Gramoli et al: “Democratizing transactional memory”
- With these additions, much greater scope possible