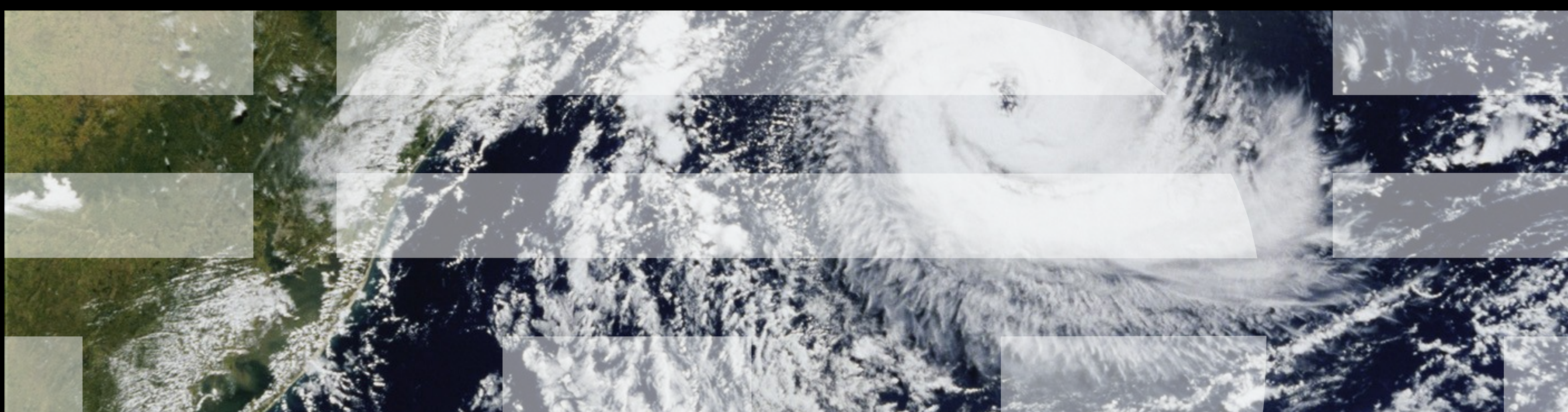
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
CPPCON, September 22, 2015

C++ Atomics: The Sad Story of memory_order_consume

A Happy Ending At Last?

(Continuation of Michael Wong's C++11/14/17 atomics talk.)



© 2015 IBM Corporation

Overview

- Target workloads
- Why memory_order_consume?
- Current sorry state of memory_order_consume in C++
- Proposed resolutions:
  - Desiderata
  - Annotating accesses
  - Annotating variables
  - Without annotation
  - Storage-class proposal
- Double-Checked Lock (if we have time)

© 2015 IBM Corporation

Target Workloads

© 2015 IBM Corporation

Target Workloads

- Workloads using linked data structures
- Balanced approach:
  - Performance must be a first-class concern …
    - If not, just write a single-threaded program and be happy
  - … but performance cannot be the only concern
    - If it was, you would be writing hand-coded assembly language
- Maximize performance while maintaining portability, maintainability, and reasonable levels of productivity
  - Goal: Effective APIs leveraging cheap hardware operations

© 2015 IBM Corporation

Why memory_order_consume?

© 2015 IBM Corporation

But First, What Is memory_order_consume?

```
atomic_store_explicit(&x, 1, memory_order_relaxed);

atomic_store_explicit(&p->a, 1, memory_order_relaxed);

atomic_store_explicit(&gp, p, memory_order_release);
```

Dependency-ordered before

Readers: Simple instructions only

```
q = atomic_load_explicit(&gp, memory_order_consume);
```

Carries a dependency

```
r1 = atomic_load_explicit(&q->a, memory_order_relaxed);

r2 = atomic_load_explicit(&x, memory_order_relaxed);
```

X

© 2015 IBM Corporation

Why The Focus On Readers?

- Today's software must adapt itself to its environment
  - Hand-built approaches unsuited today's large numbers of systems
- This environment tends to change slowly, but does change
  - The data structures representing this environment will be read-mostly
  - And they will be accessed quite frequently, as in every time that the software interacts with its environment
- Read-mostly synchronization mechanisms are thus important
  - Though there is still clearly a need for update-mostly mechanisms
  - And memory_order_consume can also be useful for updates

© 2015 IBM Corporation

Why The Focus on Eliminating Single Instructions?

- Received the following patch: saves one store & load

```
@@ -247,10 +247,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  * primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().
  */
 #define list_entry_rcu(ptr, type, member) \
-({ \
-        typeof(*ptr) __rcu *__ptr = (typeof(*ptr) __rcu __force *)ptr; \
-        container_of((typeof(ptr))rcu_dereference_raw(__ptr), type, member); \
-})
+        container_of(lockless_dereference(ptr), type, member)

 /**
  * Where are list_empty_rcu() and list_first_entry_rcu()?
```


© 2015 IBM Corporation

Why memory_order_consume?

- Received the following patch: saves one store & load
  - Both accesses non-atomic

```
@@ -247,10 +247,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
   * primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().
   */
  #define list_entry_rcu(ptr, type, member) \
 -({ \
 -         typeof(*ptr) __rcu *__ptr = (typeof(*ptr) __rcu __force *)ptr; \
 -         container_of((typeof(ptr))rcu_dereference_raw(__ptr), type, member); \
 -})
 +         container_of(lockless_dereference(ptr), type, member)

  /**
   * Where are list_empty_rcu() and list_first_entry_rcu()?
```


© 2015 IBM Corporation

Why memory_order_consume?

- Received the following patch: saves one store & load
  - Both accesses non-atomic
  - To the stack, not to a shared variable

```
@@ -247,10 +247,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  * primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().
  */
 #define list_entry_rcu(ptr, type, member) \
-({ \
-         typeof(*ptr) __rcu *__ptr = (typeof(*ptr) __rcu __force *)ptr; \
-         container_of((typeof(ptr))rcu_dereference_raw(__ptr), type, member); \
-})
+         container_of(lockless_dereference(ptr), type, member)

 /**
  * Where are list_empty_rcu() and list_first_entry_rcu()?
```


© 2015 IBM Corporation

Why memory_order_consume?

- Received the following patch: saves one store & load
  - Both accesses non-atomic
  - To the stack, not to a shared variable
  - **Some people care very deeply about performance!!!**

```
@@ -247,10 +247,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  * primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().
  */
 #define list_entry_rcu(ptr, type, member) \
-({ \
-         typeof(*ptr) __rcu *__ptr = (typeof(*ptr) __rcu __force *)ptr; \
-         container_of((typeof(ptr))rcu_dereference_raw(__ptr), type, member); \
-})
+         container_of(lockless_dereference(ptr), type, member)

 /**
  * Where are list_empty_rcu() and list_first_entry_rcu()?
```


© 2015 IBM Corporation

Why memory_order_consume?

- Received the following patch: saves one store & load
  - Both accesses non-atomic
  - To the stack, not to a shared variable
  - **Some people care very deeply about performance!!!**
    - The Linux kernel is not the only project that must accommodate their needs

```
@@ -247,10 +247,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  * primitives such as list_add_rcu() as long as it's guarded by rcu_read_lock().
  */
 #define list_entry_rcu(ptr, type, member) \
-({ \
-         typeof(*ptr) __rcu *__ptr = (typeof(*ptr) __rcu __force *)ptr; \
-         container_of((typeof(ptr))rcu_dereference_raw(__ptr), type, member); \
-})
+         container_of(lockless_dereference(ptr), type, member)

 /**
  * Where are list_empty_rcu() and list_first_entry_rcu()?
```


© 2015 IBM Corporation

Why memory_order_consume?

- Developers who face severe performance requirements:
  - Will not thank you for adding unnecessary memory-fence instructions, cache misses, or read-modify-write atomic instructions
  - Will not thank you for unnecessarily suppressing optimizations

© 2015 IBM Corporation

Why memory_order_consume?

- Developers who face severe performance requirements:
  - Will not thank you for adding unnecessary memory-fence instructions, cache misses, or read-modify-write atomic instructions
  - Will not thank you for unnecessarily suppressing optimizations
- If C and C++ are to continue to support low-level development, they must provide for extreme performance and scalability

© 2015 IBM Corporation

Why memory_order_consume?

- Developers who face severe performance requirements:
  - Will not thank you for adding unnecessary memory-fence instructions, cache misses, or read-modify-write atomic instructions
  - Will not thank you for unnecessarily suppressing optimizations
- If C and C++ are to continue to support low-level development, they must provide for extreme performance and scalability
- And this is the reason for memory_order_consume!!!
  - Intended to compile to single normal load on most CPUs
  - No atomic instructions, no memory barriers, no added overhead

© 2015 IBM Corporation

Why memory_order_consume?

- Developers who face severe performance requirements:
  - Will not thank you for adding unnecessary memory-fence instructions, cache misses, or read-modify-write atomic instructions
  - Will not thank you for unnecessarily suppressing optimizations
- If C and C++ are to continue to support low-level development, they must provide for extreme performance and scalability
- And this is the reason for memory_order_consume!!!
  - Intended to compile to single normal load on most CPUs
  - No atomic instructions, no memory barriers, no added overhead
- But how to use memory_order_consume?

© 2015 IBM Corporation

Use Case for memory_order_consume: RCU!!!

- (You can also use memory_order_consume with garbage collectors, immortal data, etc.)
- Lightest-weight conceivable read-side primitives

```
/* Assume non-preemptible (run-to-block) environment. */
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
            atomic_load_explicit(&p, memory_order_consume)
#define rcu_assign_pointer(p, v) \
            atomic_store_explicit(&p, v, memory_order_release)
```

- Results: The best possible reader performance, scalability, real-time response, wait-freedom, and energy efficiency (given good consume...)

Quick overview, references at end of slideset.

© 2015 IBM Corporation

Use Case for memory_order_consume: RCU!!!

- (You can also use memory_order_consume with garbage collectors, immortal data, etc.)
- Lightest-weight conceivable read-side primitives

```
/* Assume non-preemptible (run-to-block) environment. */
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
                atomic_load_explicit(&p, memory_order_consume)
#define rcu_assign_pointer(p, v) \
                atomic_store_explicit(&p, v, memory_order_release)
```

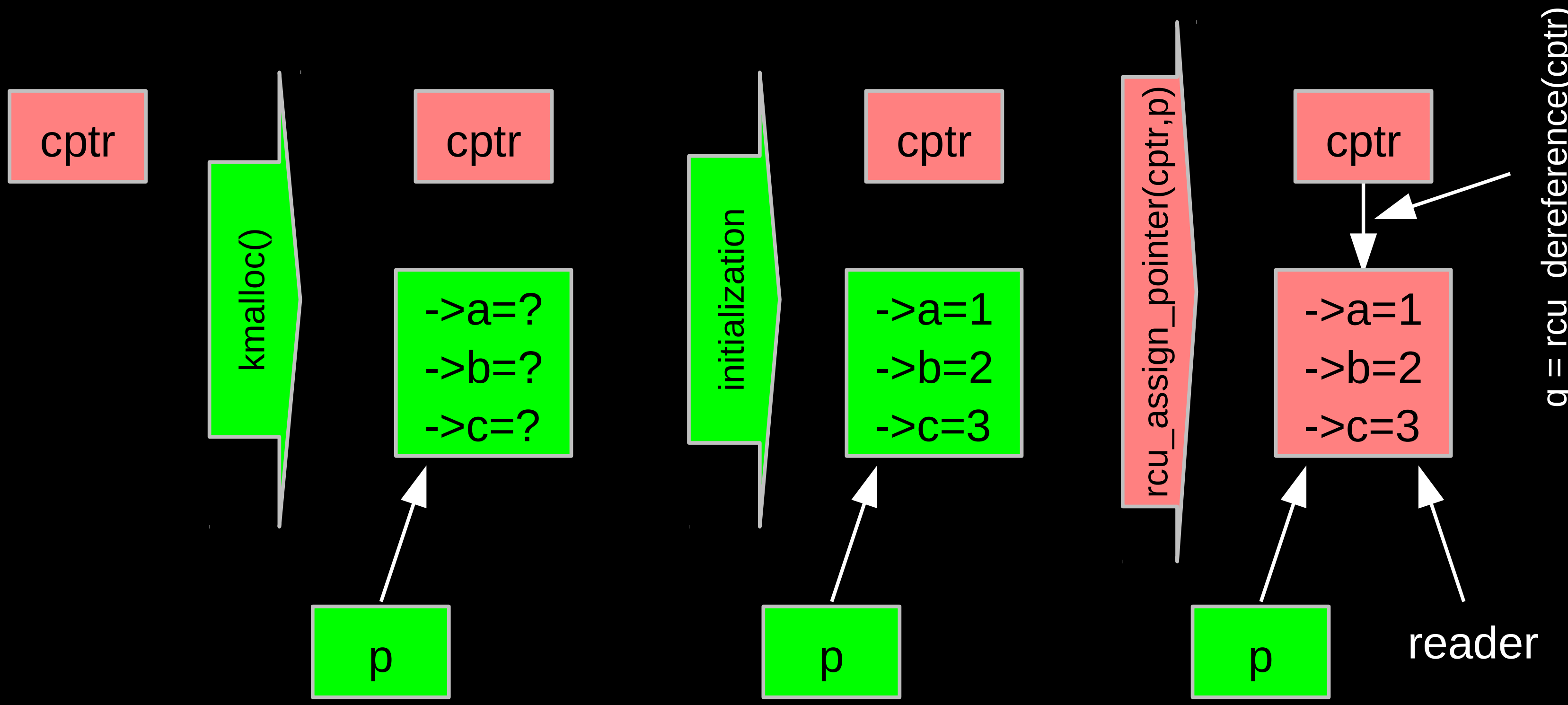
- Results: The best possible reader performance, scalability, real-time response, wait-freedom, and energy efficiency (given good consume...)
- But how can something that does not affect machine state possibly be used as a synchronization primitive???

Quick overview, references at end of slideset.

© 2015 IBM Corporation

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

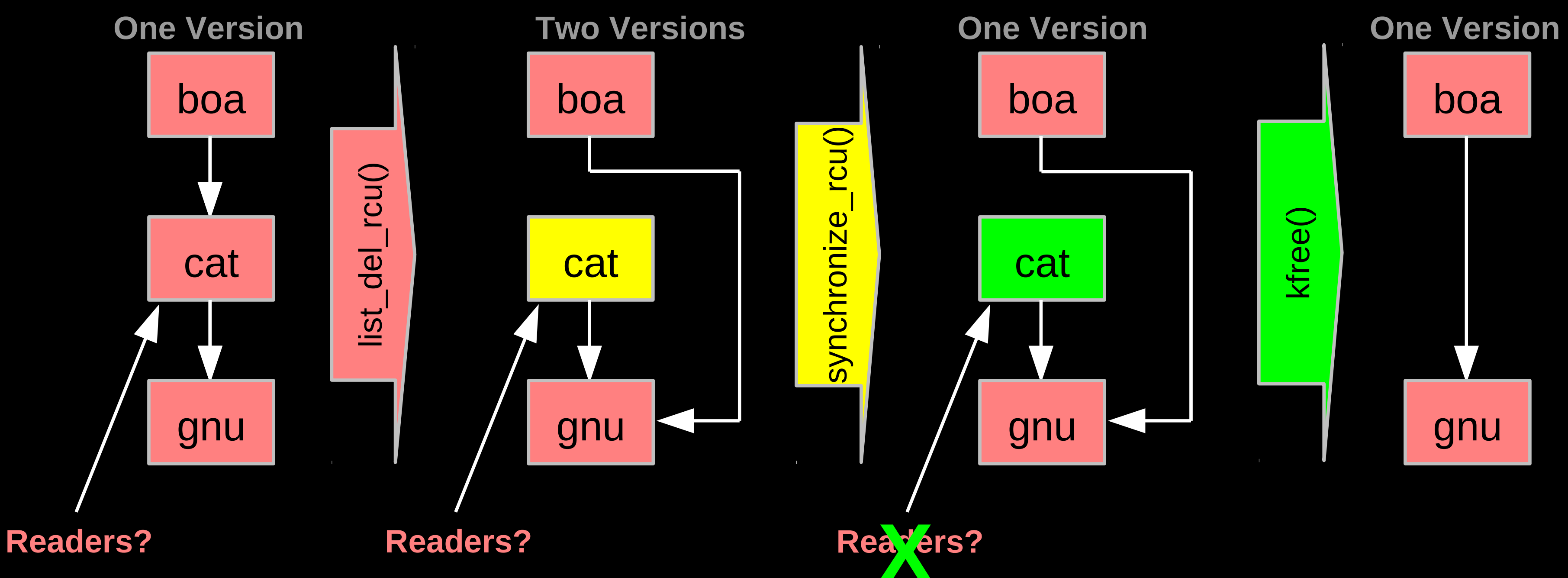


But if all we do is add, we have a big memory leak!!! (Or GC)

© 2015 IBM Corporation

RCU Safe Removal From Linked Structure

- Schroedinger's cat meets Heisenberg's uncertainty principle...
- 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())

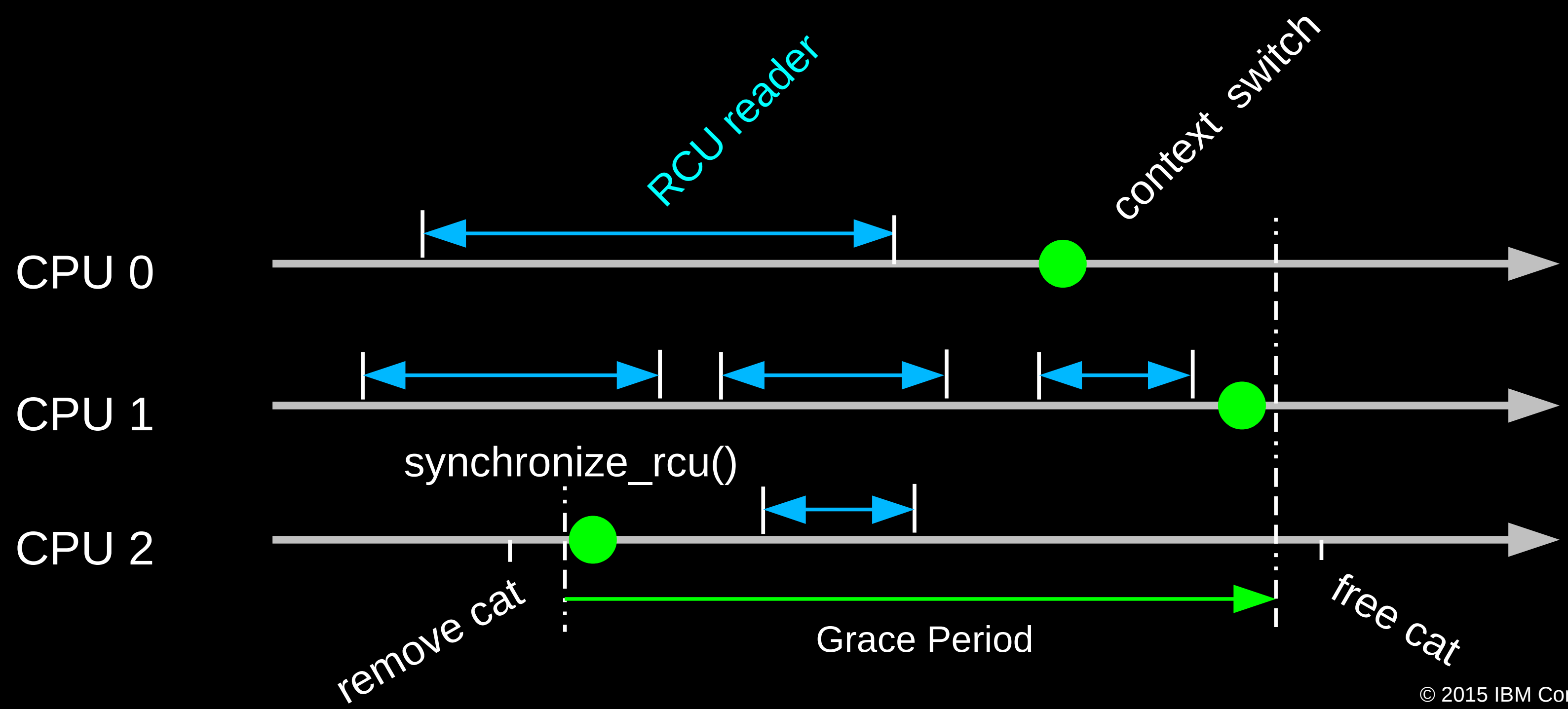


But if readers leave no trace in memory, how can we possibly tell when they are done???

© 2015 IBM Corporation

RCU Waiting for Pre-Existing Readers: Quiescent State-Based Reclamation (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



© 2015 IBM Corporation

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

© 2015 IBM Corporation

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***

© 2015 IBM Corporation

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 some RCU implementations do use lightweight code in addition to social engineering

© 2015 IBM Corporation

RCU Avoids Contention and Expensive Instructions

Want to be here!

16-CPU 2.8GHz Intel X5550 (Nehalem) System

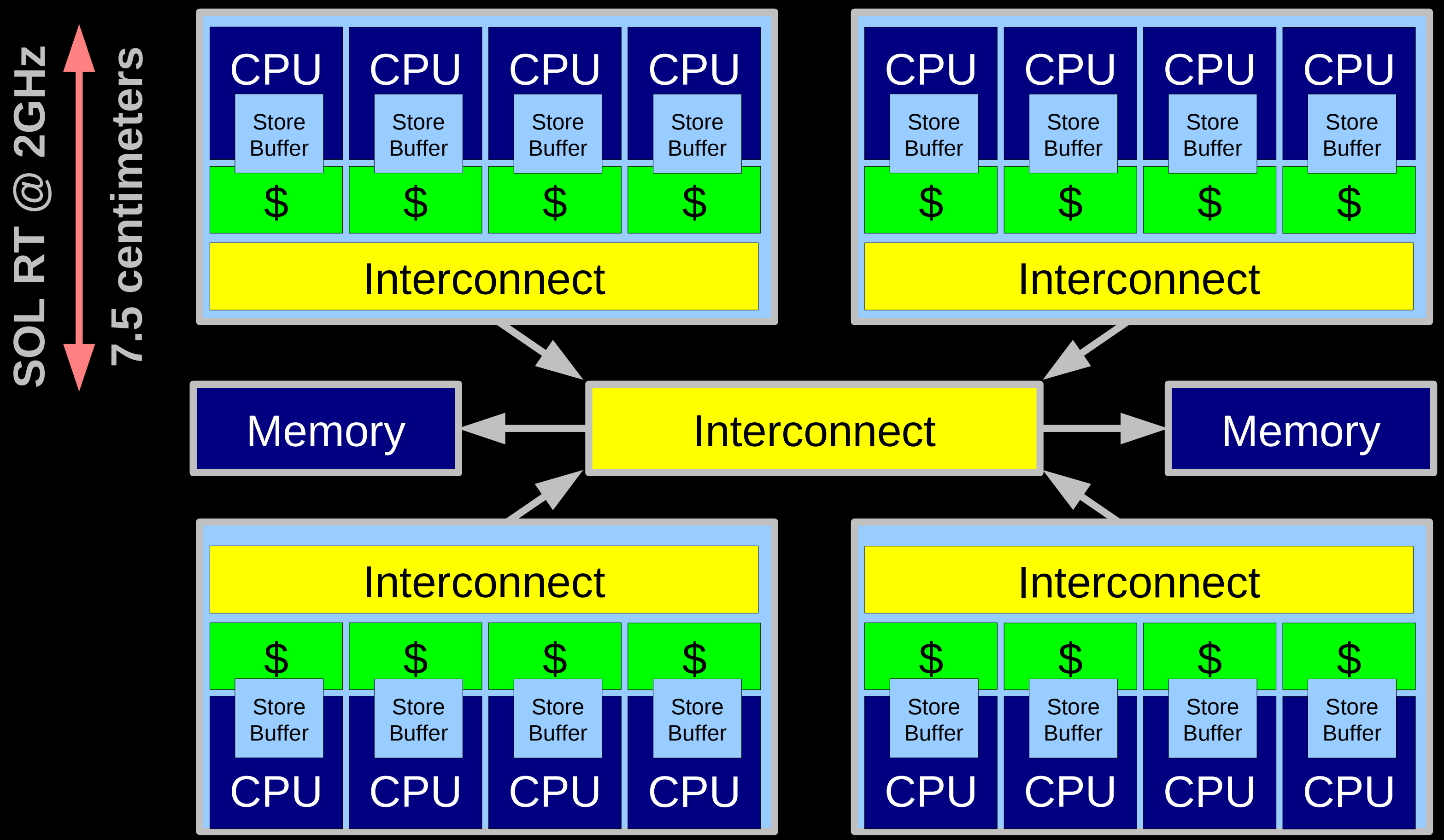
Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**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

© 2015 IBM Corporation

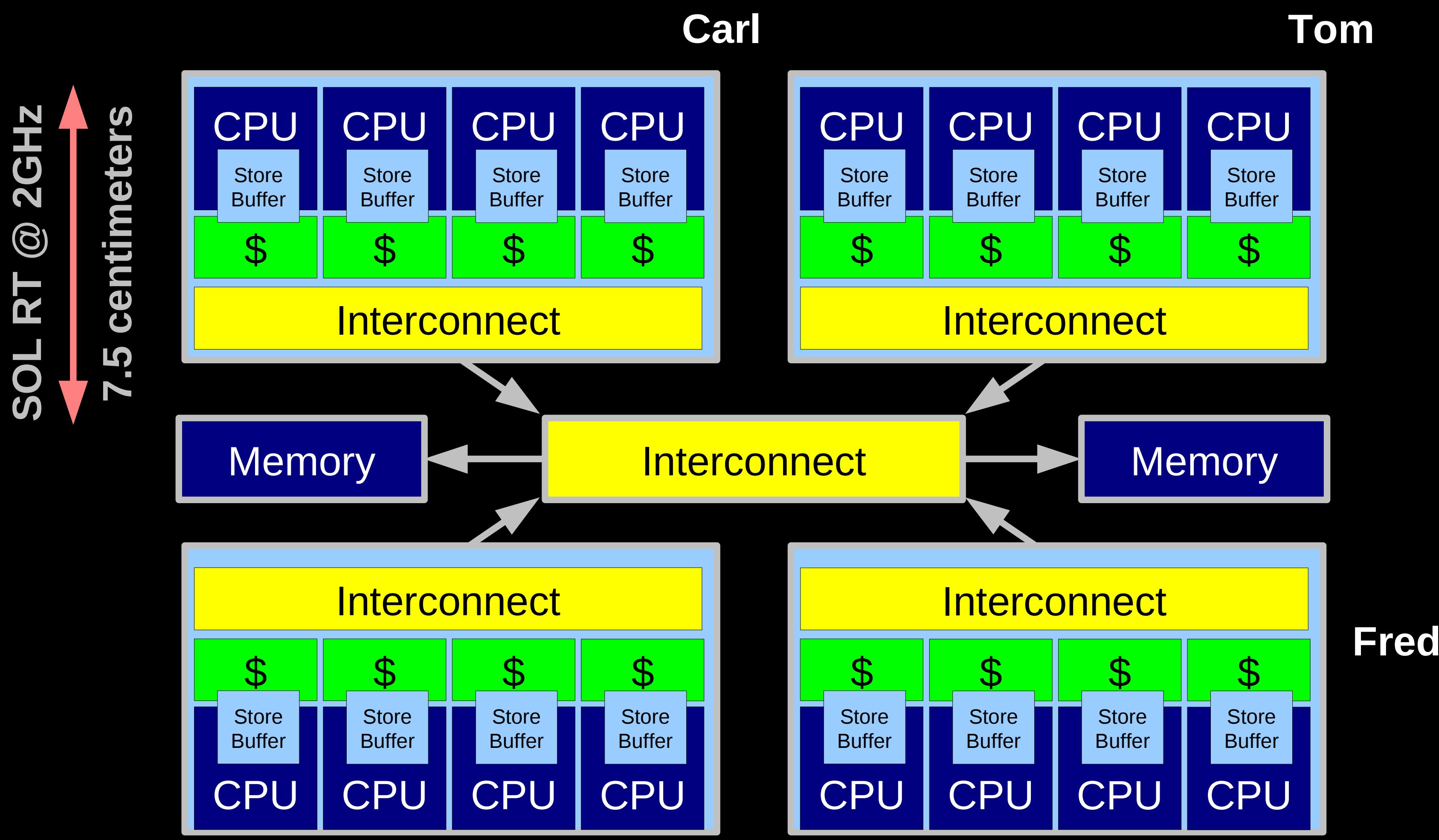
Hardware Structure



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

© 2015 IBM Corporation

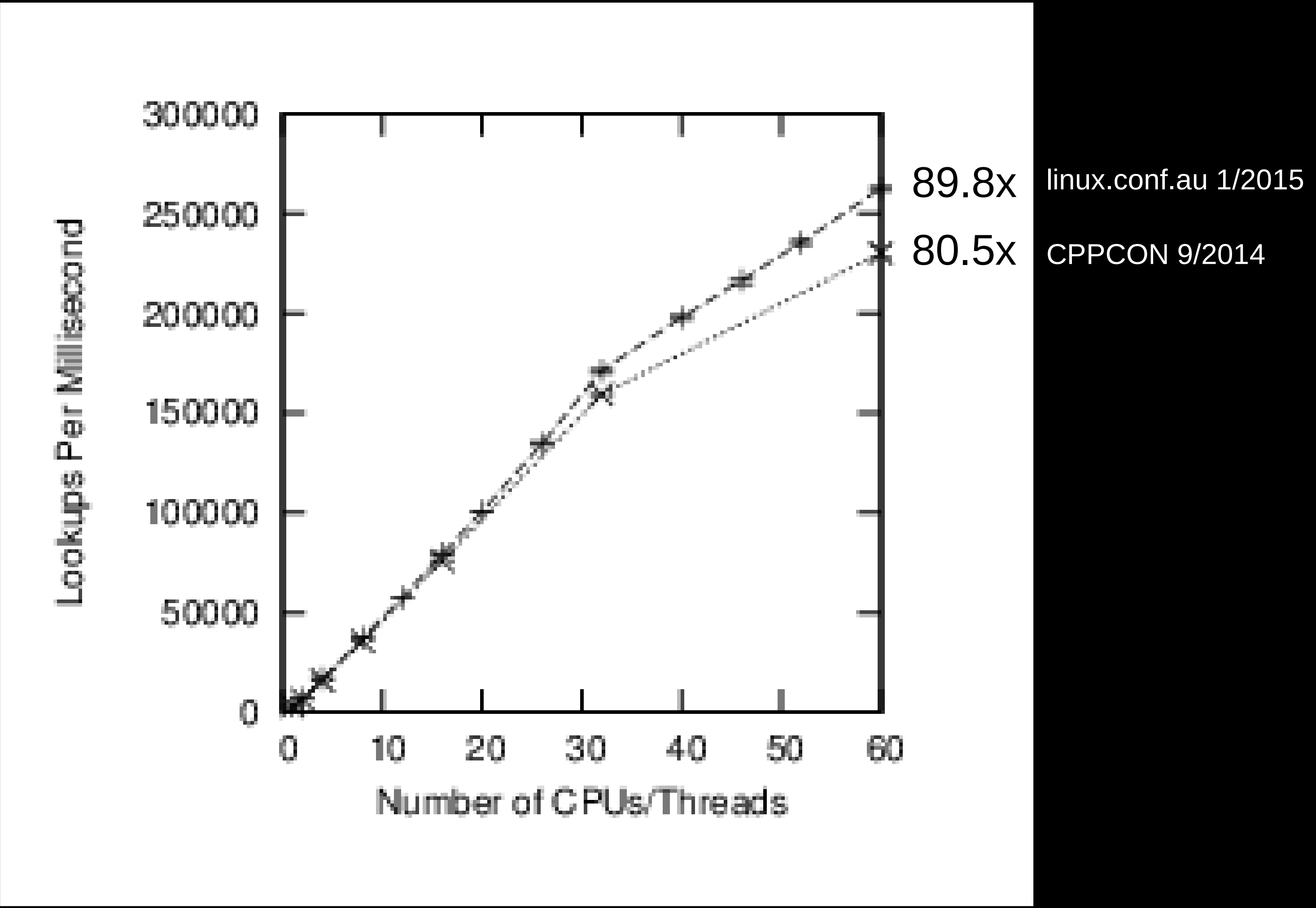
Hardware Structure



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

© 2015 IBM Corporation

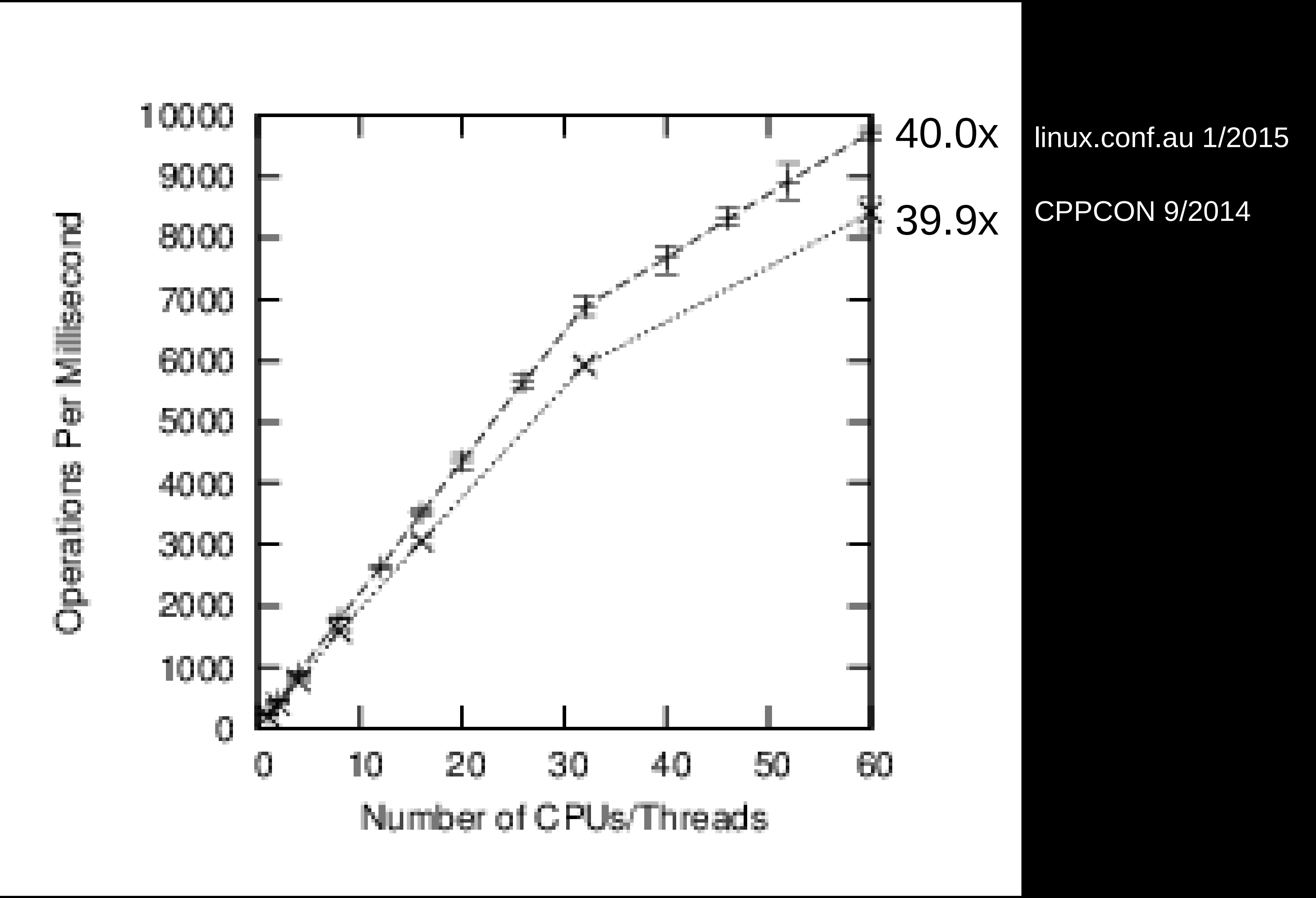
RCU's Binary Search Tree Read-Only Performance



100% lookups
Super-linear as expected based on range partitioning
(Hash tables about 3x faster)

© 2015 IBM Corporation

RCU: Binary Search Tree Mixed Performance



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

© 2015 IBM Corporation

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

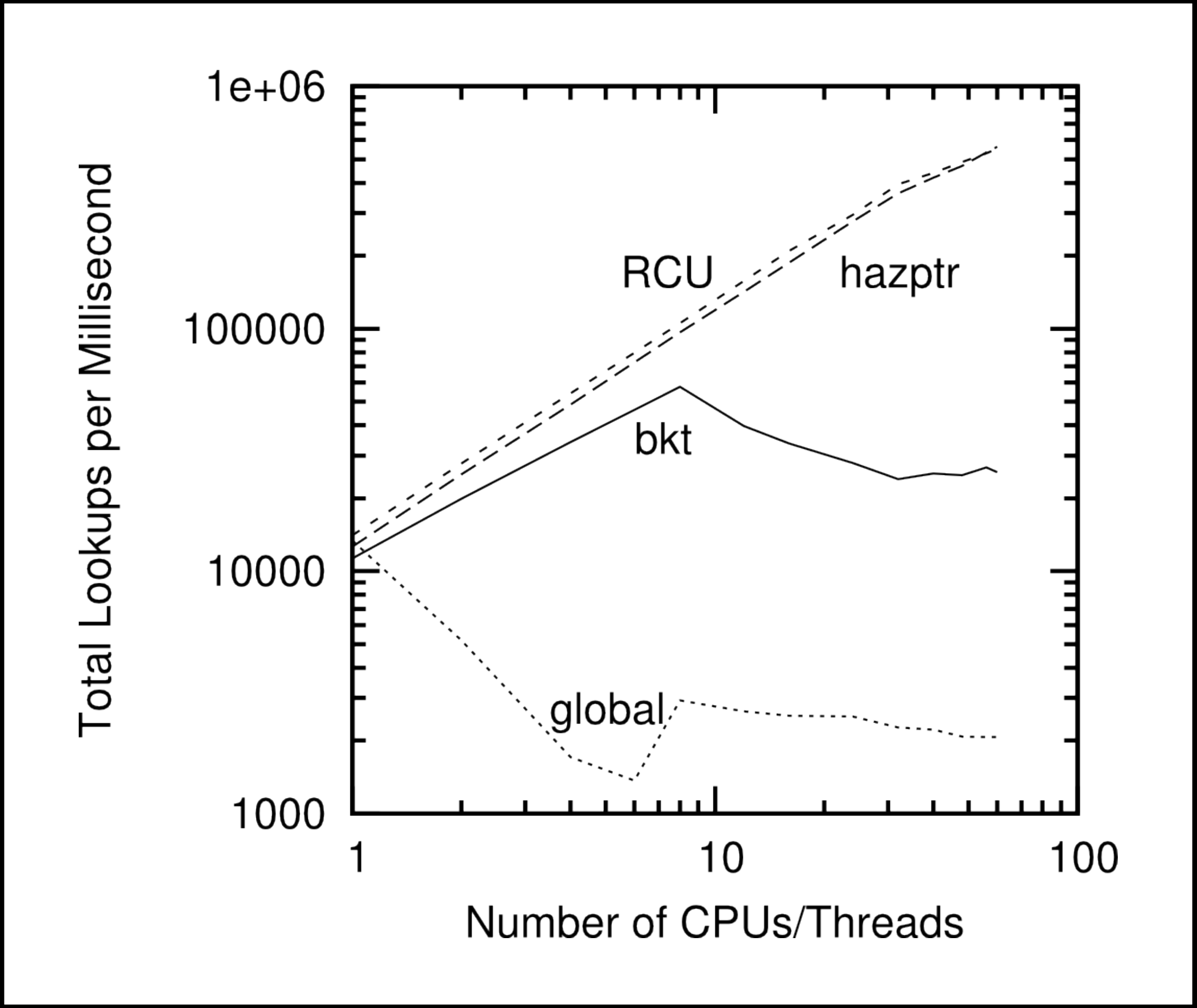
```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); \
        (p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

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

5 of which might someday be replaced by memory_order_consume

© 2015 IBM Corporation

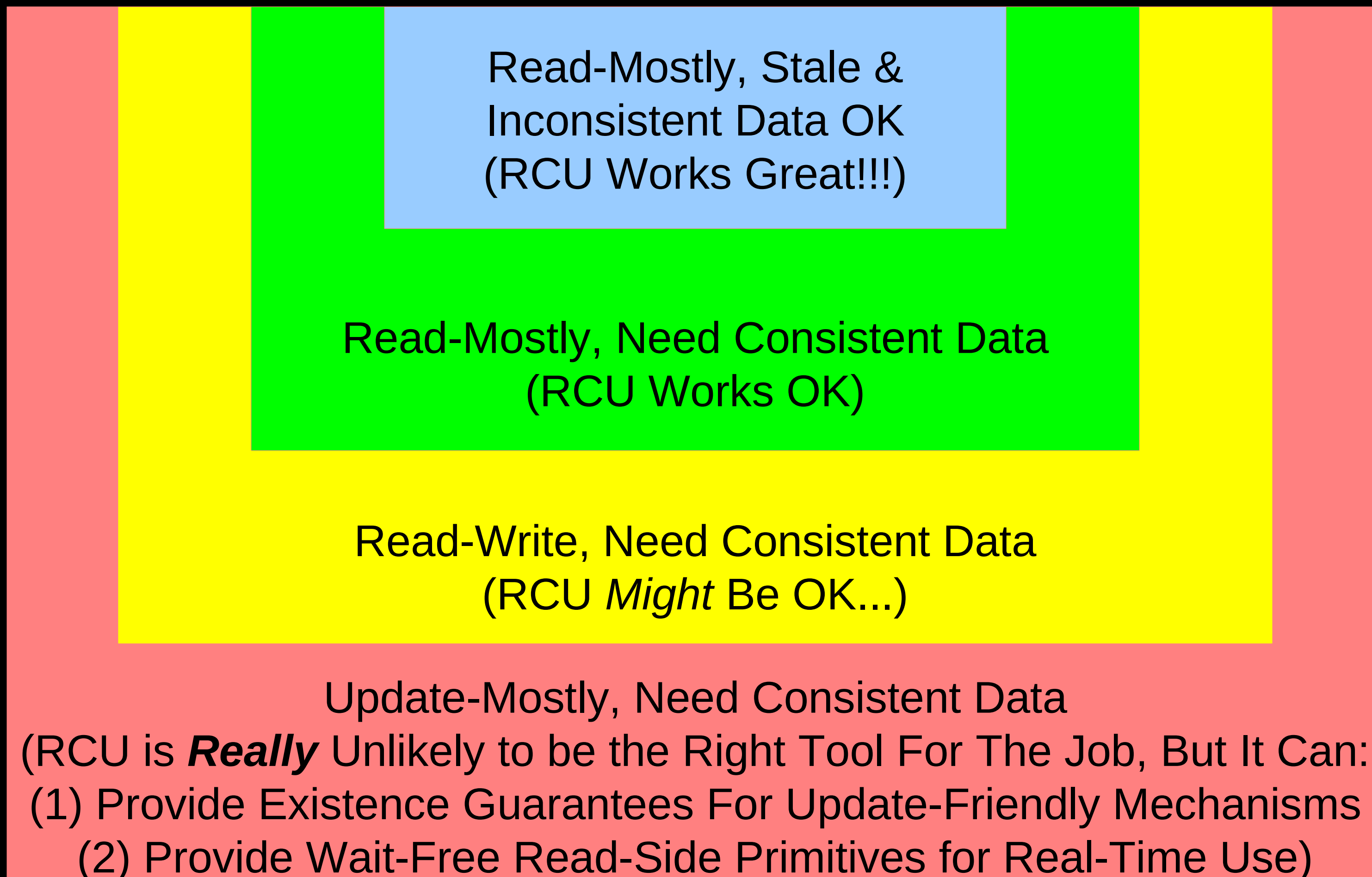
RCU Performance: Read-Only Hash Table



RCU and hazard pointers scale quite well!!!

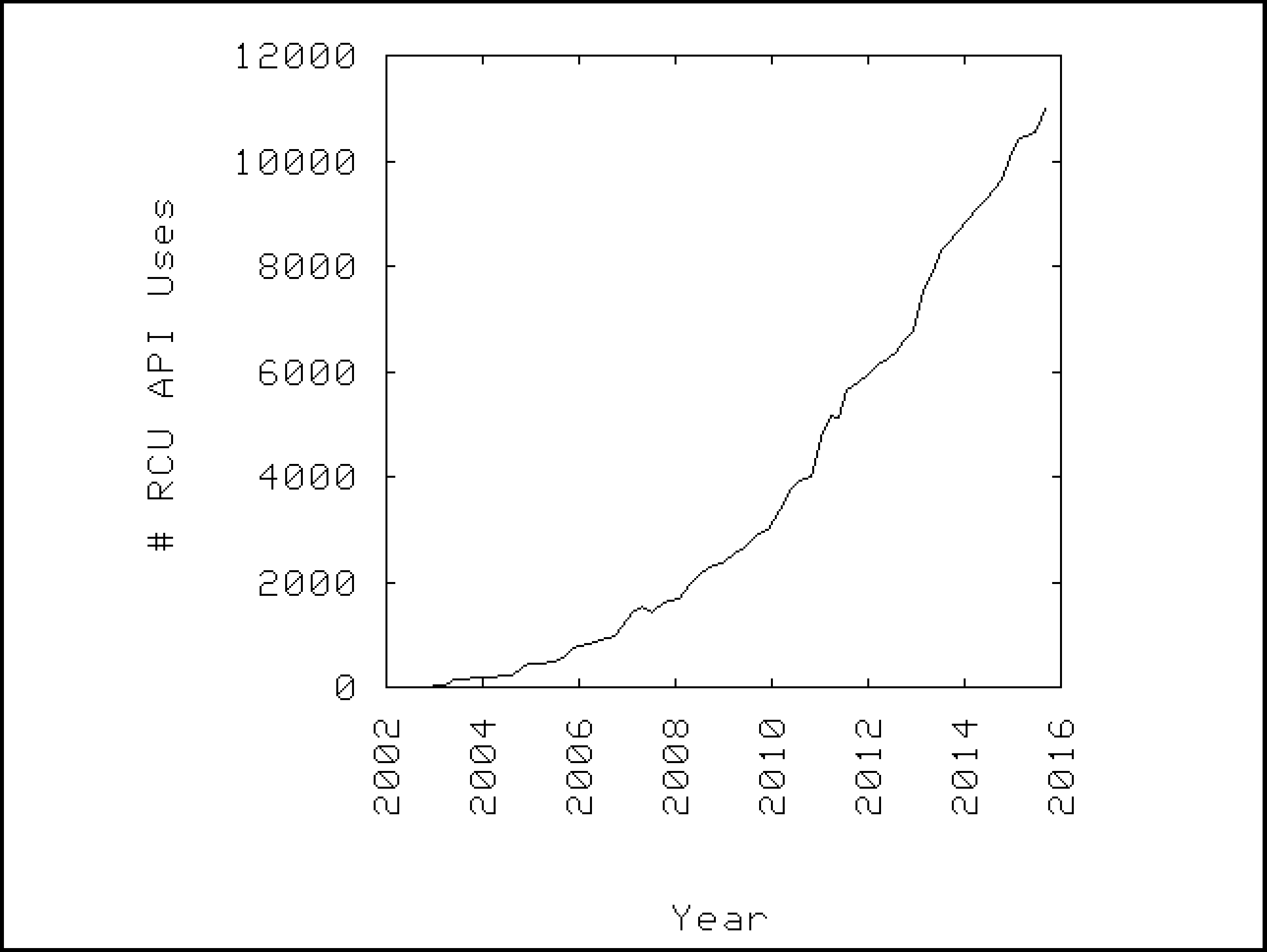
© 2015 IBM Corporation

RCU Area of Applicability



© 2015 IBM Corporation

RCU Applicability to the Linux Kernel



This records the community's work, not mine!

© 2015 IBM Corporation

Benefits of RCU, Where Applicable

- Fast and scalable readers
  - “Free is a very good price” and “Nothing is faster than doing nothing”
  - RCU usage has resulted in order-of-magnitude speedups
- Wait-free readers eliminates many forms of deadlock
  - Can't deadlock without waiting
  - First use in DYNIX/ptx eliminated 16KLoC of subtle code
- Retry-free readers eliminates many forms of livelock
  - Can't livelock without retries
- Wait-free and retry-free readers well-suited to real-time
- Eliminates ABA storage-reuse problem
  - “Poor person's garbage collector”
- Plays well with other synchronization primitives

© 2015 IBM Corporation

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); /* External function! */
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
- Dependency chains can and do fan in (see above), fan out, and cross compilation-unit boundaries (see above)

© 2015 IBM Corporation

RCU Usage: Dependency Chains Can Fan Out

- This happens when abstracting data-structure access:

```
struct foo *get_rcu_ref(void)
{
    return rcu_dereference(cptr);
}

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


© 2015 IBM Corporation

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

© 2015 IBM Corporation

RCU Usage: kill_dependency() Use Case

- kill_dependency(): Hand off from RCU to other mechanism

```
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr);
if (nlt = need_long_term(p)) {
    atomic_inc(&p->refcount);
    p = kill_dependency(p);
}
rcu_read_unlock(); /* End critical section. */
if (nlt)
    do_something_longterm(p);
else
    /* *p might be freed!!! */
```

- Can also hand off to locks, hazard pointers, etc.

© 2015 IBM Corporation

Current Sorry C++ State of memory_order_consume

© 2015 IBM Corporation

Current Sorry C++ State of memory_order_consume

- An evaluation A *carries a dependency* to an evaluation B if
  - The value of A is used as an operand of B, unless:
    - B is an invocation of any specialization of std::kill_dependency (29.3), or
    - A is the left operand of a built-in logical AND (&&, see 5.14) or logical OR (||, see 5.15) operator, or
    - A is the left operand of a conditional (?:, see 5.16) operator, or
    - A is the left operand of the built-in comma (,) operator (5.18):
  - or
    - A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or
    - for some evaluation X, A carries a dependency to X, and X carries a dependency to B
  - [*Note:* “Carries a dependency to” is a subset of “is sequenced before', and is similarly strictly intra-thread. – *end note*]

© 2015 IBM Corporation

Current Sorry C++ State of memory_order_consume

- An evaluation A *carries a dependency* to an evaluation B if
  - The value of A is used as an operand of B, unless:
    - B is an invocation of any specialization of std::kill_dependency (29.3), or
    - A is the left operand of a built-in logical AND (&&, see 5.14) or logical OR (||, see 5.15) operator, or
    - A is the left operand of a conditional (?:, see 5.16) operator, or
    - A is the left operand of the built-in comma (,) operator (5.18):
  - or
    - A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or
    - for some evaluation X, A carries a dependency to X, and X carries a dependency to B
  - [*Note:* “Carries a dependency to” is a subset of “is sequenced before', and is similarly strictly intra-thread. – *end note*]
- Current compilers simply promote to memory_order_acquire
  - Resulting in memory-fence instructions and suppressed optimizations
  - And failing to suppress read-fusion optimizations...

© 2015 IBM Corporation

What memory_order_consume Taught Me

- I have also learned a lot about RCU in the meantime
  - In 1999: About 100 uses of RCU in DYNIX/ptx
  - In 2006: About 1,000 RCU uses in the Linux kernel
  - In 2015: More than 10,000 RCU uses in the Linux kernel
- And memory_order_consume has severe usability problems:
  - Need explicit kill_dependency() to terminate chain
    - Forgetting one of them silently provides you a costly memory fence
  - Need [[carries_dependency]] attribute for external functions
    - Without this, compilers must emit memory fences at function calls
  - Limited ability to issue diagnostics for common usage errors
    - Probably need warning on each memory fence emitted for dependency
  - Arbitrary integer computations difficult to deal with
    - A smart compiler will break dependencies to insert known constants
    - Which is a good thing, even in concurrent programs
    - But without memory-barrier instructions

© 2015 IBM Corporation

What memory_order_consume Taught Me

- The three things that compiler writers hate most are:
  - Tracing dependency chains

© 2015 IBM Corporation

What memory_order_consume Taught Me

- The three things that compiler writers hate most are:
  - Tracing dependency chains
  - *Tracing dependency chains*

© 2015 IBM Corporation

What memory_order_consume Taught Me

- The three things that compiler writers hate most are:
  - Tracing dependency chains
  - *Tracing dependency chains*
  - ***Tracing dependency chains***
- Small wonder consume just gets promoted to acquire!!!
- But there are important use cases needing a high-quality memory_order_consume implementation
  - Current volatile-cast work-arounds are sort of OK, but we really need something much better

© 2015 IBM Corporation

Proposed Resolutions

http://www.rdrop.com/users/paulmck/submission/consume.2015.09.22a.pdf

© 2015 IBM Corporation

Proposed Resolutions: Desiderata

- Easily evaluated dependency type
  - Avoid schemes requiring compiler to trace dependencies
- Easily specified dependencies
  - Avoid attributes for C compatibility (or C can use keyword)
  - Enable abstraction aligned with current compiler practice
  - Near term, Linux kernel compatibility (implementation experience!)
  - Long term, enable high-quality diagnostics
- Avoid unsolicited memory-barrier instructions
  - The point of all this is to ***increase*** performance and predictability
- Tractable to modern formal-verification methods

© 2015 IBM Corporation

Proposed Resolutions: Types of Dependency Chains

- Strict dependency (dep):
  - Purely syntactic, as in C++11, and the only one that is easy to model
- Semantic dependency (sdep):
  - Chain is broken if only one value is possible anywhere in the chain
- Local semantic dependency (lsdep):
  - Chain is broken if only one value is possible anywhere in the chain, ignoring the possibility that only one value might be loaded by the memory_order_consume load heading the chain
    - The compiler must assume that the initial load can return any value in its type even if it knows better
- Restricted dependency (rdep):
  - Chain is maintained only by selected pointer operations
    - As in those used on Linux-kernel dependency chains
  - Chain is broken if compiler can see that only one value is possible anywhere in the chain

© 2015 IBM Corporation

Examples of Dependency Chains (1/4)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);  /* Which uses q->a, q->b, etc. */
rcu_read_unlock();
```

- dep: Dependency chain persists
- sdep: Dependency chain persists
- lsdep: Dependency chain persists
- rdep: Dependency chain persists

© 2015 IBM Corporation

Examples of Dependency Chains (2/4)

- Compiler can guess pointer value

```
initialize(&mystruct);
rcu_assign_pointer(gp, &mystruct); /* Only assignment in program! */
…
rcu_read_lock();
q = rcu_dereference(gp);
if (q)
        do_something_with(q);  /* Compiler knows q == &mystruct */
rcu_read_unlock();
```

- dep: Dependency chain persists (memory fence?)
- sdep: Dependency chain broken by smart compiler
- lsdep: Dependency chain persists (memory fence?)
- rdep: Dependency chain broken by smart compiler

© 2015 IBM Corporation

Examples of Dependency Chains (3/4)

- Compiler can guess pointer value, take 2

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
rcu_read_lock();
q = rcu_dereference(gp);
if (q == cached_p)
      do_something_with(q);  /* Compiler knows q == cached_p */
rcu_read_unlock();
```

- dep: Dependency chain persists (memory fence?)
- sdep: Dependency chain broken, even by stupid compiler
- lsdep: Dependency chain broken, even by stupid compiler
- rdep: Dependency chain broken, even by stupid compiler

© 2015 IBM Corporation

Examples of Dependency Chains (4/4)

- Dependency carried through an integer

```
initialize(&x[i]);
rcu_assign_pointer(gx, i); /* Many assignments, no guessing */
…
rcu_read_lock();
i = rcu_dereference(gx);
if (i >= 1 && i < MAX_IDX)
        do_something_with(&x[i - 1]);  /* Compiler knows nothing */
rcu_read_unlock();
```

- dep: Dependency chain persists
- sdep: Dependency chain persists
- lsdep: Dependency chain persists
- rdep: Dependency chain broken (in theory)

© 2015 IBM Corporation

Proposed Resolutions List

1) Annotating accesses
2) Annotating variables
3) No annotations
4) Storage class

© 2015 IBM Corporation

Proposed Resolution 1: Annotating Accesses

© 2015 IBM Corporation

Proposed Resolution 1: Annotating Accesses

- Explicitly tail-marked dependency chains (dep, Section 7.7)
- Explicitly head-marked dependency chains (dep, Section 7.8)
  - Both suggested by Olivier Giroux

© 2015 IBM Corporation

Tail-Marked Access Annotations (Section 7.7)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(atomic_dependency(q, gp));
rcu_read_unlock();
```

- Must enforce dependency ordering, using fences if needed

© 2015 IBM Corporation

Head-Marked Access Annotations (Section 7.8)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
rcu_read_lock();
q = rcu_dereference(gp, q);
do_something_with(q);
rcu_read_unlock();
```

- Must enforce dependency ordering, using fences if needed

© 2015 IBM Corporation

Annotating Accesses: Summary

- Explicitly tail-marked dependency chains (Section 7.7)
- Explicitly head-marked dependency chains (Section 7.8)
  - Some compiler implementers *really* like these
  - Seems to require tracing dependency chains, though through binary
  - Emits unsolicited memory-fence instructions
    - Lots of them if dependency chain passes through many translation units
  - Not clear that this supports modularity
    - How far does dependency chain extend? Fan in? Fan out?
    - Perhaps mark formal and actual parameters to extend in and return type to extend out
  - Additional refinement quite possible
    - The text was generated from very vague descriptions

© 2015 IBM Corporation

Proposed Resolution 2: Annotating Variables

© 2015 IBM Corporation

Proposed Resolution 2: Annotating Variables

- Type-based designation of dependency chains with restrictions (lsdep, Section 7.2)
  - Suggested by Torvald Riegel
- Type-based designation of dependency chains (dep, Section 7.3)
  - Suggested by Jeff Preshing
- Mark dependency-carrying local variables (dep, Section 7.6)
  - Suggested by Clark Nelson

© 2015 IBM Corporation

Type-Based Designation of Dependency Chains With Restrictions (Section 7.2)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
struct foo value_dep_preserving *q;
void do_something_with(struct foo value_dep_preserving *p);
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);
rcu_read_unlock();
```

- Semantic dependency: No unsolicited memory fences?
- Assignments to/from value_dep_preserving variables?

© 2015 IBM Corporation

Type-Based Designation of Dependency Chains (Section 7.3)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
struct foo value_dep_preserving *q;
void do_something_with(struct foo value_dep_preserving *p);
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);
rcu_read_unlock();
```

- Strict dependency: Unsolicited memory fences (diagnostic?)
- Assignments to/from value_dep_preserving variables?

© 2015 IBM Corporation

Mark Dependency-Carrying Local Variables (Section 7.6)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
struct foo [[carries_dependency]] *q;
void do_something_with(struct foo [[carries_dependency]] *p);
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);
rcu_read_unlock();
```

- Strict dependency, but only via some operations
- Assigning to unattributed variable kills dependency
- C11 doesn't do attributes, so use keyword instead for C

© 2015 IBM Corporation

Annotating Variables: Summary

- Type-based designation of dependency chains with restrictions (Section 7.2)
  - Modifying the type system is a big ask
- Type-based designation of dependency chains (Section 7.3)
  - Modifying the type system is again a big ask
- Mark dependency-carrying local variables (Section 7.6)
  - Might work longer term given variable modifier instead of attribute
    - As suggested Lawrence Crowl (see later slides)
  - Also need formal parameters, actual parameters, and return values
  - Might work well for new code base, but not for today's Linux kernel

© 2015 IBM Corporation

Proposed Resolution 3: Without Annotations

© 2015 IBM Corporation

Proposed Resolution 3: Without Annotations

- Whole-program option (sdep, Section 7.4)
  - Suggested by Jeff Preshing
- Local-variable restriction (dep?, Section 7.5)
  - Suggested by Hans Boehm
- Restricted dependency chains (rdep, Section 7.9)
  - Suggested by yours truly

© 2015 IBM Corporation

Without Annotations Means Without Annotations!

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);
rcu_read_unlock();
```

- Much cleaner source code, no unsolicited fences
- Much more difficult to produce diagnostics and formal tools

© 2015 IBM Corporation

Without Annotations: Summary

- Whole-program option (Section 7.4)
  - Refined as “restricted dependency chains” below
- Local-variable restriction (Section 7.5)
  - Comes close, but gives unsolicited memory-fence instructions
  - Also refined as “restricted dependency chains” below
- Restricted dependency chains (Section 7.9)
  - “Just say no!” to carrying dependencies through integer computations
    - Suitable for large existing code bases
  - Compiler less likely to break pointer-based dependency chains
    - This proposal codifies pointer-based dependency chains
    - Longer term, variable marking can provide improved diagnostics and bring formal-verification tools back into the picture

© 2015 IBM Corporation

Proposed Resolution 4: Storage Class (Section 7.10)

- Common case in Linux kernel code

```
initialize(p);  /* Dynamically allocated. */
rcu_assign_pointer(gp, p); /* Many assignments, no guessing */
…
_Carries_dependency struct foo *q;
void do_something_with(_Carries_dependency struct foo *p);
…
rcu_read_lock();
q = rcu_dereference(gp);
do_something_with(q);
rcu_read_unlock();
```

- Strict dependency, but only via operations called out in standard
  - And only on pointer types: intptr_t, and uintptr_t limited as in 7.9
  - _Carries_dependency cannot be applied to other types
- Assigning to non-_Carries_dependency variable kills dependency
  - Considered C++ attribute, but need to change semantics
- Should work well for formal methods

© 2015 IBM Corporation

_Carries_dependency Interactions

- `_Carries_dependency`: Object carries a dependency
- `register _Carries_dependency`: Register variable carries a dependency
- `static _Carries_dependency`: static variable carries a dependency
- `static thread_local _Carries_dependency`: static thread-local variable carries a dependency
- `extern _Carries_dependency`: external thread-local variable carries a dependency
- `extern thread_local _Carries_dependency`: external thread-local variable carries a dependency
- `thread_local _Carries_dependency`: thread-local variable carries a dependency

© 2015 IBM Corporation

Storage Class: Summary (Section 7.10)

- No need to trace dependencies
- Dependency chains pruned by default when assigning to non-_Carries_dependency objects
- No need for attributes in C
- No modifications to the type system
- Not a short-term solution for the Linux kernel
- Should enable analysis tools based on formal methods

© 2015 IBM Corporation

Double-Checked Lock

© 2015 IBM Corporation

Double-Checked Lock: Reader

- (Hey, Fedor started this!)
- Have pointer be flag, avoiding need to synchronize them
  - Dependency ordering will provide this order for free
- Enclose check in RCU read-side critical section
  - This makes it easy to determine when to free old structure
- Use usermode RCU
  - So it is OK to block in RCU read-side critical sections
  - Solution is a bit more ornate in the Linux kernel
- Untested, probably does not even compile
  - Bonus points for bugs spotted

© 2015 IBM Corporation

Double-Checked Lock: Reader

```
rcu_read_lock();
p = rcu_dereference(gp);  /* memory_order_consume */
if (!p) {
    mutex_lock(&gp_lock);
    p = rcu_dereference(gp);
    if (!p) {
        p = malloc(sizeof(*p));
        if (!p)
            handle_oom();  /* Does not return. */
        initialize(p);
        rcu_assign_pointer(gp, p);
    }
    mutex_unlock(&gp_lock);
}
do_something(p);
rcu_read_unlock();
```


© 2015 IBM Corporation

Double-Checked Lock: Updater

```
if (need_change()) {
    p = NULL;
    mutex_lock(&gp_lock);
    if (need_change()) {
        p = rcu_dereference(gp);
        rcu_assign_pointer(gp, NULL);  /* Next reader allocates. */
    }
    mutex_unlock(gp_lock);
    if (p) {
        synchronize_rcu();
        kfree(p);
    }
}
```


© 2015 IBM Corporation

Summary and Conclusions

© 2015 IBM Corporation

Summary and Conclusions

- Happy ending at last?

© 2015 IBM Corporation

Summary and Conclusions

- Happy ending at last? Maybe!

© 2015 IBM Corporation

Summary and Conclusions

- Happy ending at last? Maybe!
  - Restricted dependency chains (Section 7.9) for existing code bases
    - Some dispute as to whether or not this requires standardization
  - Storage class (Section 7.10) for new projects
    - Hopefully existing projects migrate in this direction
- But very early days for these two proposals
  - So watch this space!!!

© 2015 IBM Corporation

To Probe Deeper (RCU)

- https://queue.acm.org/detail.cfm?id=2488549
  – "Structured Deferral: Synchronization via Procrastination" (also in July 2013 CACM)
- http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159 and http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf
  – "User-Level Implementations of Read-Copy Update"
- git://lttng.org/userspace-rcu.git (User-space RCU git tree)
- http://people.csail.mit.edu/nickolai/papers/clements-bonsai.pdf
  – Applying RCU and weighted-balance tree to Linux mmap_sem.
- http://www.usenix.org/event/atc11/tech/final_files/Triplett.pdf
  – RCU-protected resizable hash tables, both in kernel and user space
- http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf
  – Combining RCU and software transactional memory
- http://wiki.cs.pdx.edu/rp/: Relativistic programming, a generalization of RCU
- http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/
  – "What is RCU?" Series
- http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf
  – RCU motivation, implementations, usage patterns, performance (micro+sys)
- http://www.livejournal.com/users/james_morris/2153.html
  – System-level performance for SELinux workload: >500x improvement
- http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf
  – Comparison of RCU and NBS (later appeared in JPDC)
- http://doi.acm.org/10.1145/1400097.1400099
  – History of RCU in Linux (Linux changed RCU more than vice versa)
- http://read.seas.harvard.edu/cs261/2011/rcu.html
  – Harvard University class notes on RCU (Courtesy of Eddie Koher)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)

© 2015 IBM Corporation

To Probe Deeper (1/5)

- Hash tables:
  - http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.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

© 2015 IBM Corporation

To Probe Deeper (2/5)

- 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 &c: “Using Read-Copy-Update Techniques for System V IPC in 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/
  - McKenney: “High-Performance and Scalable Updates: The Issaquah Challenge”
    - http://www2.rdrop.com/users/paulmck/scalability/paper/Updates.2015.01.16b.LCA.pdf
    - (Update to 2014 CPPCON presentation)

© 2015 IBM Corporation

To Probe Deeper (3/5)

- 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

© 2015 IBM Corporation

To Probe Deeper (4/5)

- 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/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/
  - McKenney: RCU requirements series: http://lwn.net/Articles/652156/, http://lwn.net/Articles/652677/, http://lwn.net/Articles/653326/
- 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
  - McKenney: “High-Performance and Scalable Updates: The Issaquah Challenge”
    - http://www2.rdrop.com/users/paulmck/scalability/paper/Updates.2015.01.16b.LCA.pdf

© 2015 IBM Corporation

To Probe Deeper (5/5)

- RCU theory and semantics, academic contributions (partial list)
  - Gamsa et al., “Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System”
    - http://www.usenix.org/events/osdi99/full_papers/gamsa/gamsa.pdf
  - McKenney, “Exploiting Deferred Destruction: An Analysis of RCU Techniques”
    - http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf
  - Hart, “Applying Lock-free Techniques to the Linux Kernel”
    - http://www.cs.toronto.edu/~tomhart/masters_thesis.html
  - Olsson et al., “TRASH: A dynamic LC-trie and hash data structure”
    - http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4281239
  - Desnoyers, “Low-Impact Operating System Tracing”
    - http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf
  - Dalton, “The Design and Implementation of Dynamic Information Flow Tracking ...”
    - http://csl.stanford.edu/~christos/publications/2009.michael_dalton.phd_thesis.pdf
  - Gotsman et al., “Verifying Highly Concurrent Algorithms with Grace (extended version)”
    - http://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf
  - Liu et al., “Mindicators: A Scalable Approach to Quiescence”
    - http://dx.doi.org/10.1109/ICDCS.2013.39
  - Tu et al., “Speedy Transactions in Multicore In-memory Databases”
    - http://doi.acm.org/10.1145/2517349.2522713
  - Arbel et al., “Concurrent Updates with RCU: Search Tree as an Example”
    - http://www.cs.technion.ac.il/~mayaarl/podc047f.pdf

© 2015 IBM Corporation

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.

© 2015 IBM Corporation

Questions?

© 2015 IBM Corporation