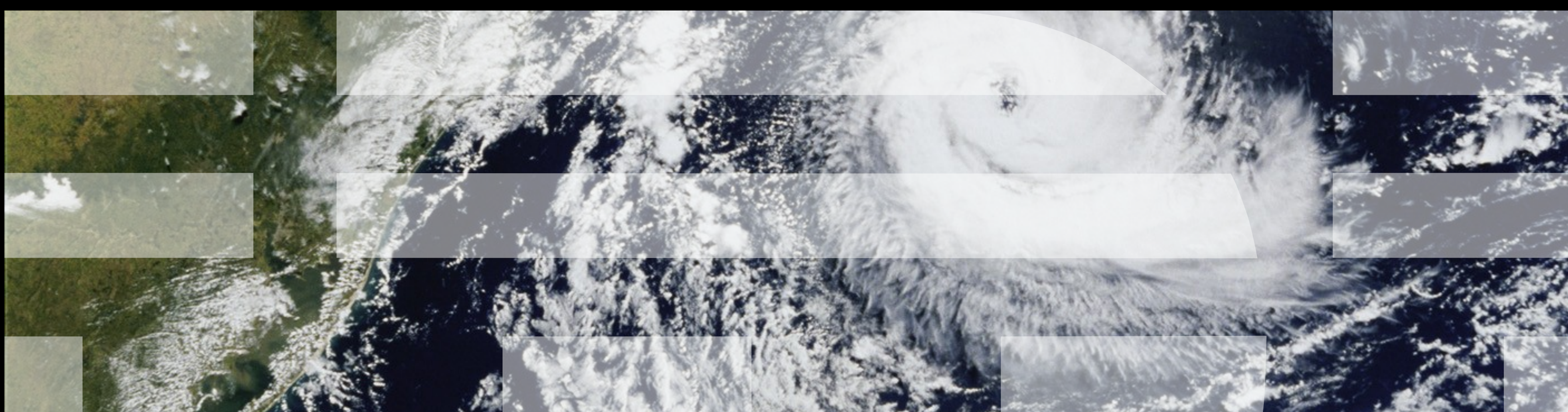
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
Reorder Workshop, Vienna, Austria, July 17, 2014

Reordering and Verification in the Linux Kernel



Overview

- Linux Kernel and Weak Ordering
- What Is RCU?
- Linux Kernel Validation: A Grand Challenge
- Linux Kernel Validation State of the Art and Mitigations
- Linux Kernel Validation: Future Possibilities

Linux Kernel and Weak Ordering

Linux Kernel and Weak Ordering

- Split counters
  - Each CPU increments is own counter to update, occasional statistical readout sums all CPUs' counters: No ordering required
- Memory allocator
  - Fastpath has neither atomic instructions or memory barriers
  - However, there are kfree()-to-kmalloc() requirements across CPUs
- RCU
  - More on this in the following slides...

Linux Kernel and Weak Ordering

- Split counters
  - Each CPU increments is own counter to update, occasional statistical readout sums all CPUs' counters: No ordering required
- Memory allocator
  - Fastpath has neither atomic instructions or memory barriers
  - However, there are kfree()-to-kmalloc() requirements across CPUs
- RCU
  - More on this in the following slides...

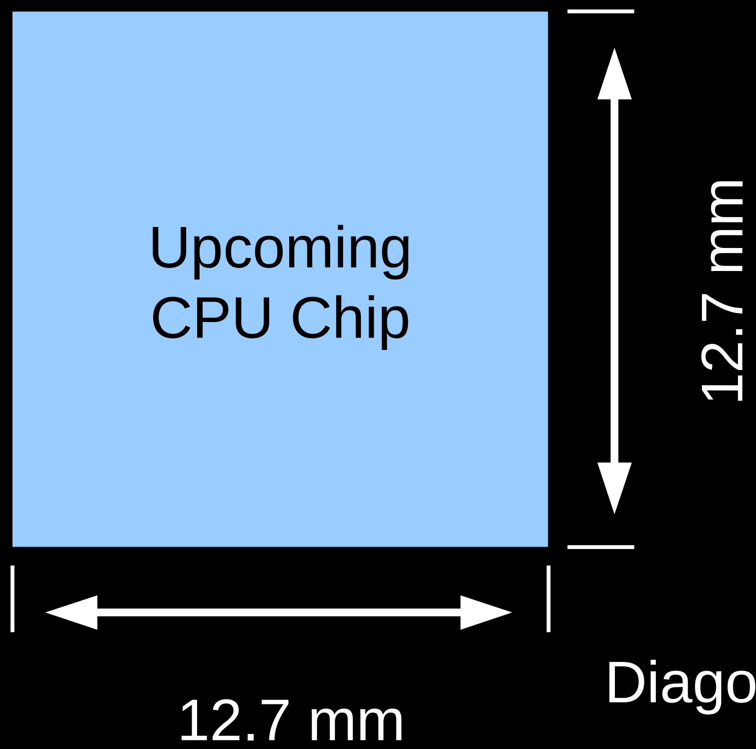
- Lots of opportunity for reordering in the Linux kernel!!!

What Is RCU?

Why RCU?

- To accommodate the laws of physics
  - And other trivial issues...

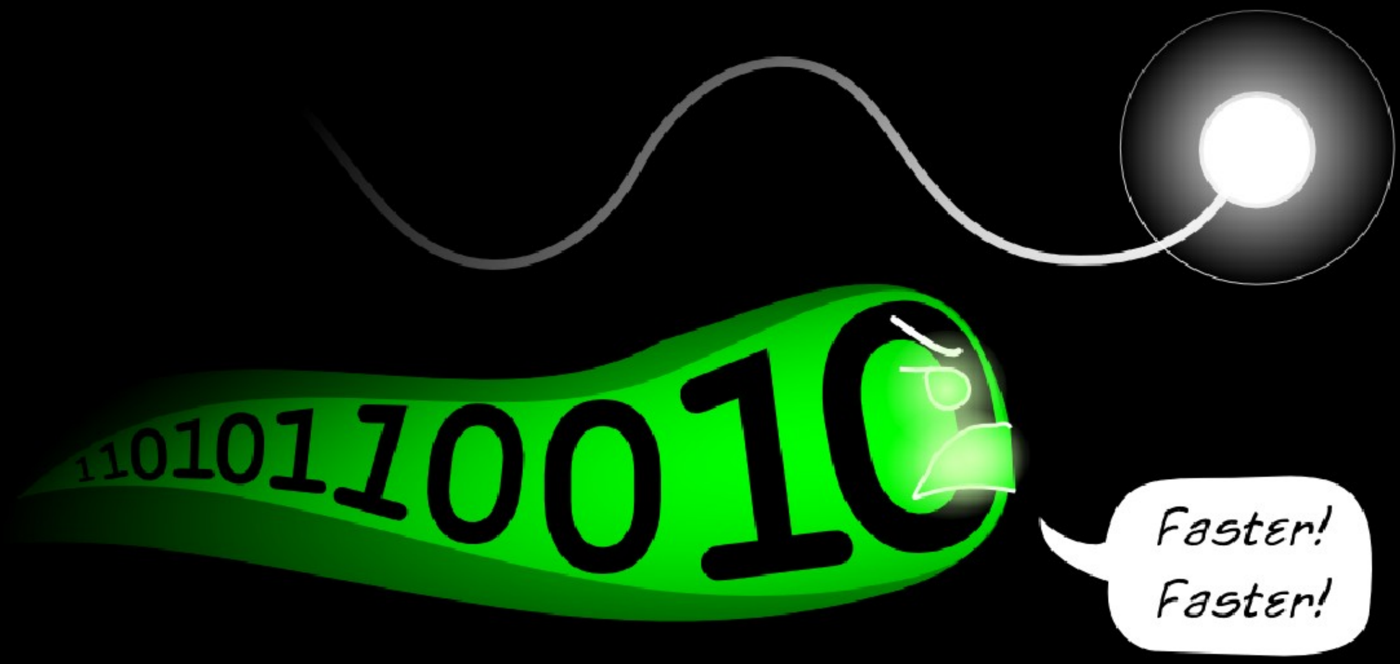
Speed of Light (to Say Nothing of Electrons) is Finite; Size of Computers is Non-Zero



Diagonally across chip and back (35.8mm):
3.6 clocks at 1GHz
17.9 clocks at 5GHz
Out for the request, back to return the data

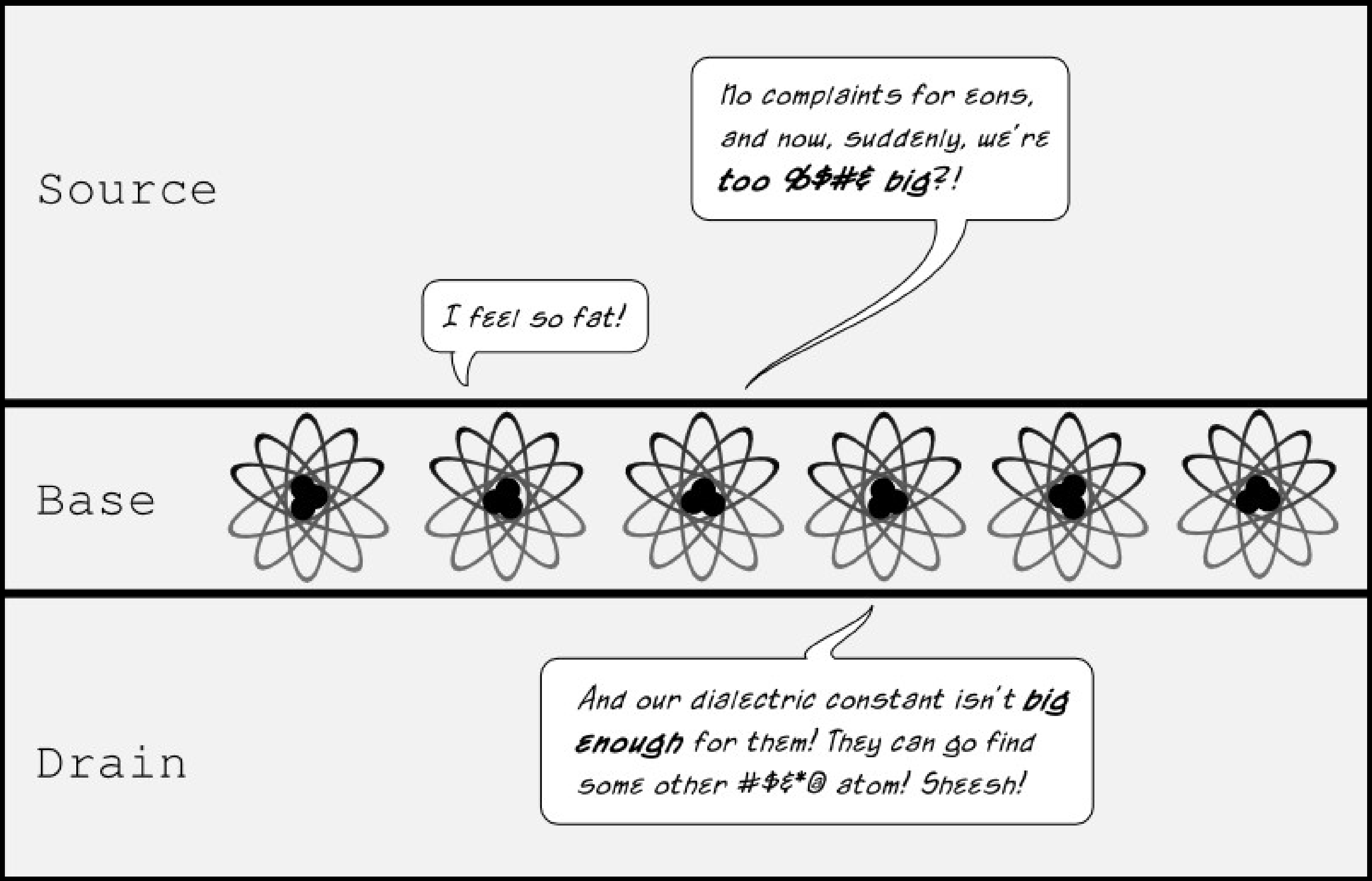
Source: http://en.wikipedia.org/wiki/List_of_upcoming_intel_processors

Problem With Physics #1: Finite Speed of Light



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

Problem With Physics #2: Atomic Nature of Matter



Performance of Synchronization Mechanisms

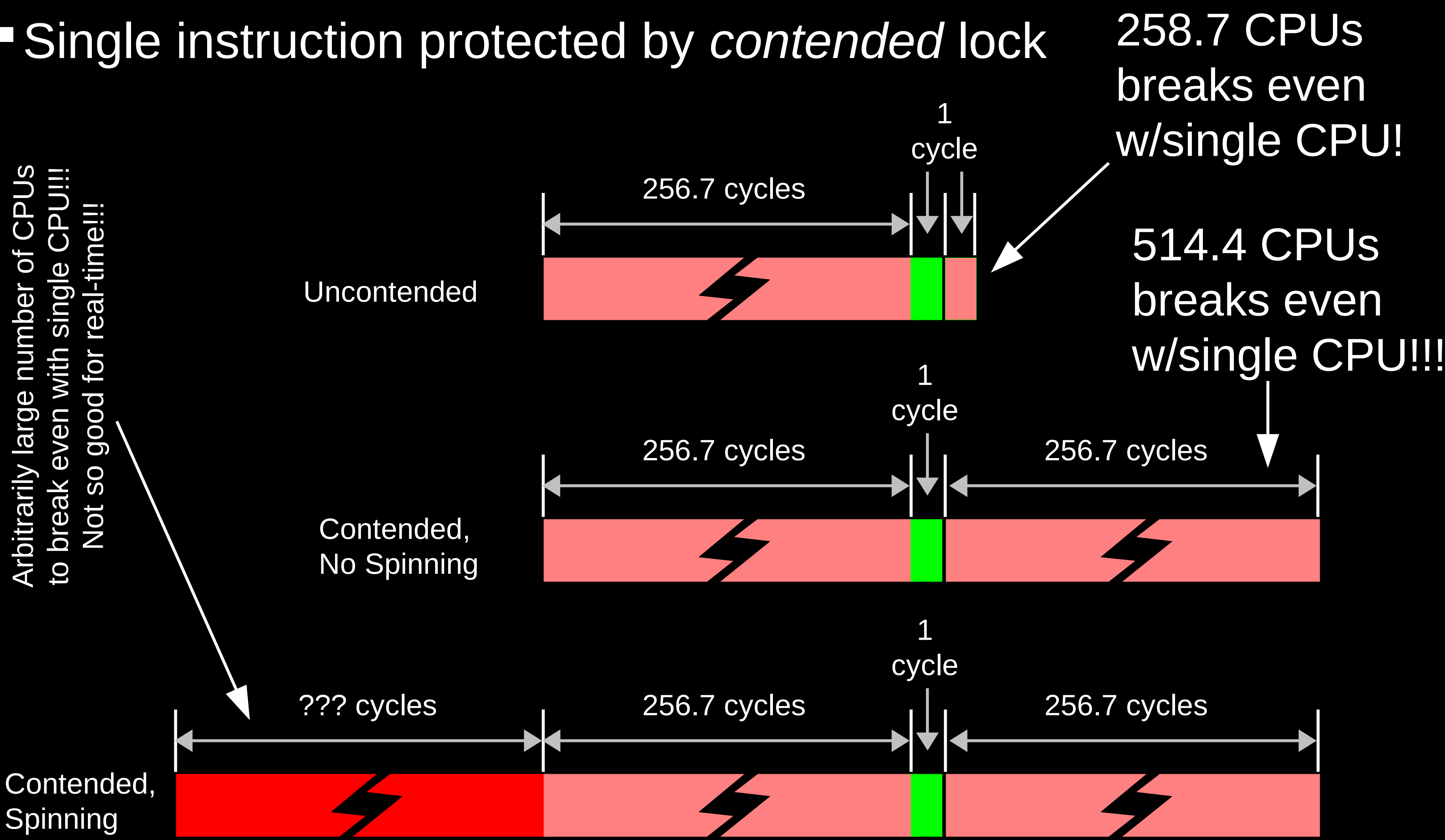
16-CPU 2.8GHz Intel X5550 (Nehalem) System

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

That 3.6 and 17.9 clocks now looks pretty good...
Buffering, queueing and caching result in substantial additional performance degradation!

But What Do The Operation Timings Really Mean???

- Single instruction protected by *contended* lock



Also Applies to Reader-Writer Locking, Non-Blocking Synchronization and Transactional Memory

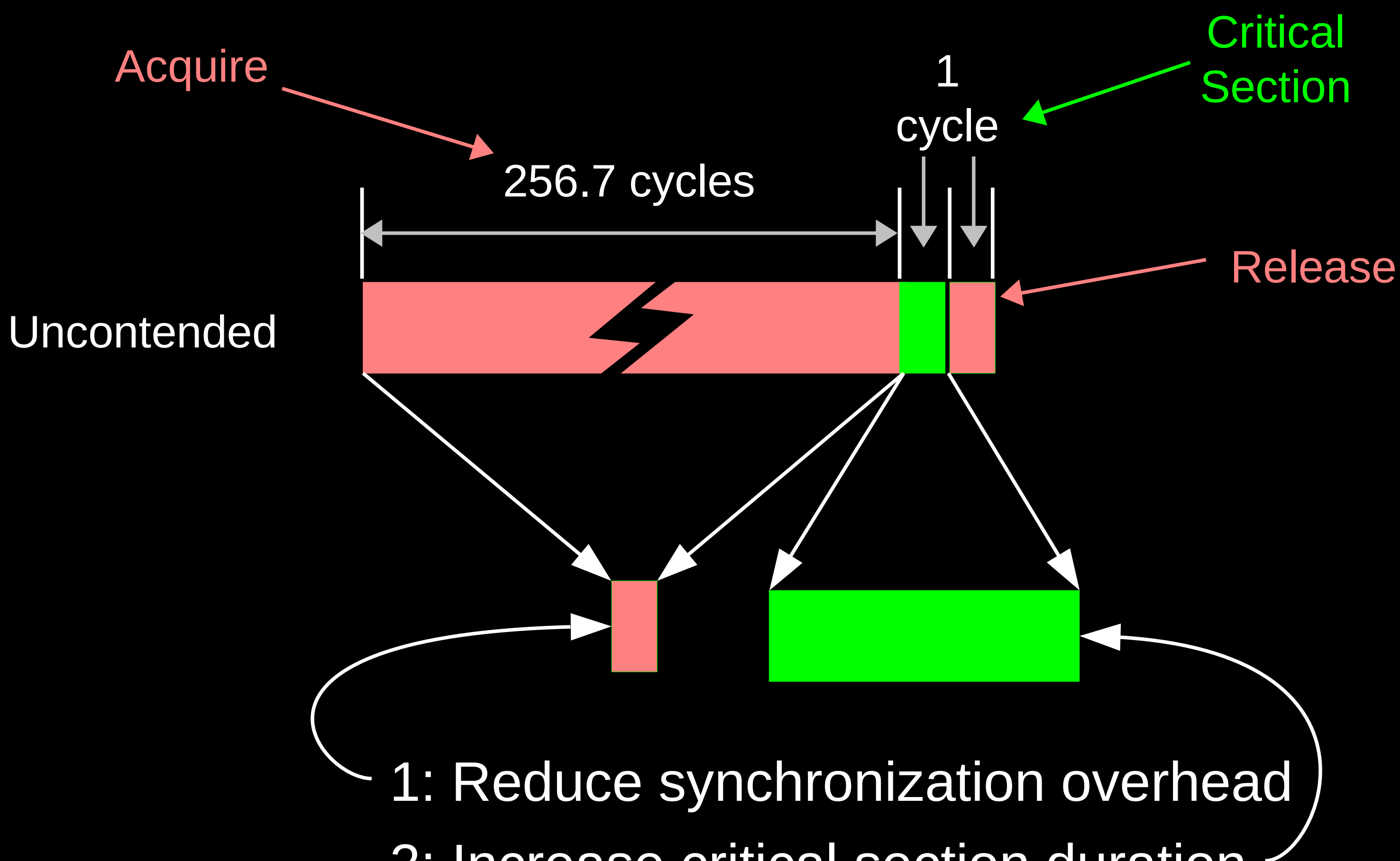
Though read-only transactions can be heavily optimized, but not as heavily as RCU can.

Can't Hardware Do Better Than This???

- There might be some ways to improve hardware:
  - 3D lithography: Too bad about power and heat dissipation!
  - Extreme ultraviolet lithography: Making progress, but limited
  - Liquid immersion lithography: Making progress, but limited
  - Asynchronous logic: big in the '60s, starting to be used again
  - Exotic materials (e.g., graphene): Promising, but still a research toy
  - Light rather than electrons: Promising, but still a research toy
  - Vacuum-channel transistors: Promising, but still a research toy
  - ***Wormholes: Works great on Star Trek!!!***
  - ***Hyperspace: Works great on Star Wars!!!***
- Although hardware will continue to improve, software needs to do its part: “Free lunch” exponential performance improvement of 80s and 90s is over

How Can Software Live With This Hardware???

Two Basic Ways To Proceed...



1: Reduce synchronization overhead

2: Increase critical section duration

We will focus on option #1, for readers.
(In real life, you need to do both.)

Design Principle: Avoid Expensive Operations

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Use cheap-and-cheerful operations

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

Taking It To The Limit...

"Only those who have gone too far
can possibly tell you how far you can go!!!"

Taking It To The Limit...

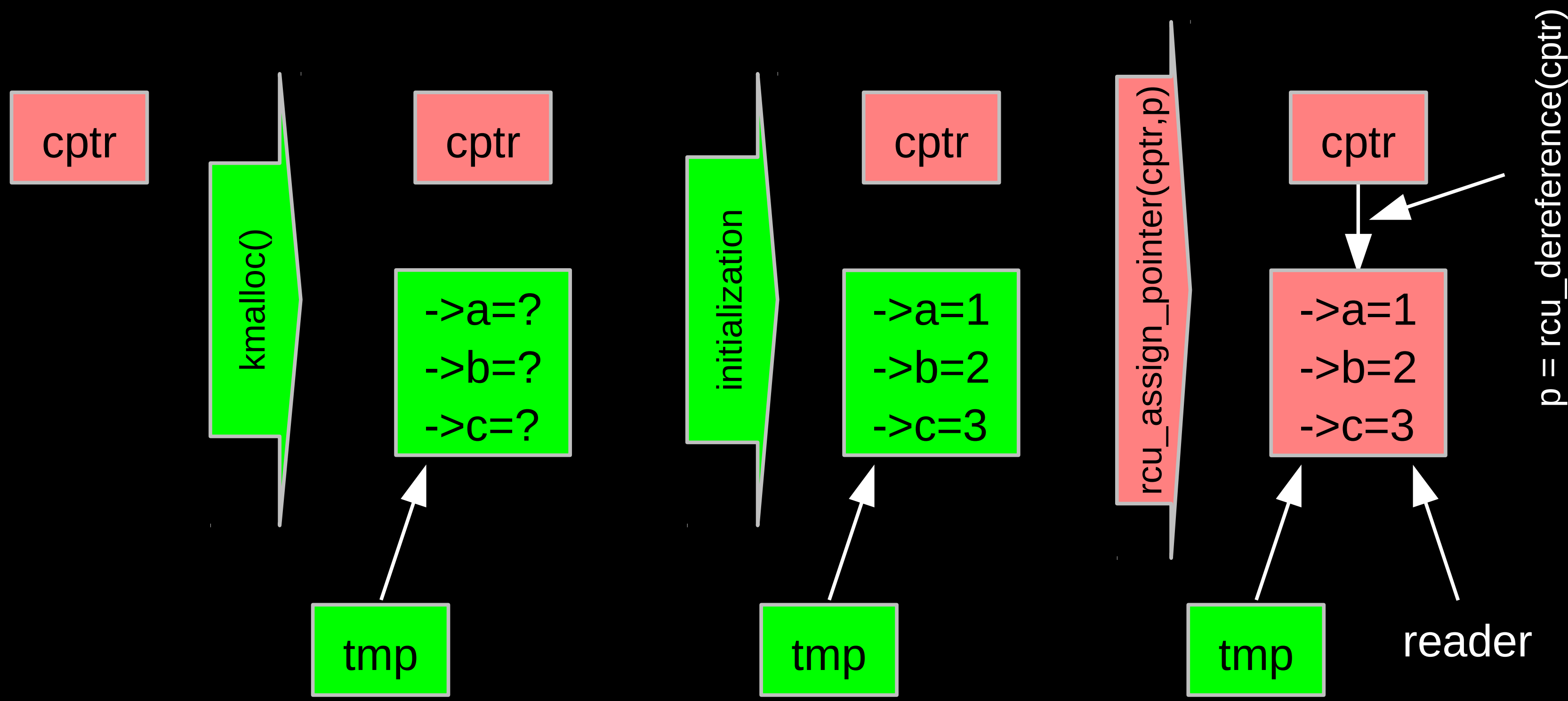
- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency

Taking It To The Limit...

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency
- But how can a primitive that doesn't affect machine state possibly be a useful synchronization primitive?

Publication of And Subscription to New Data

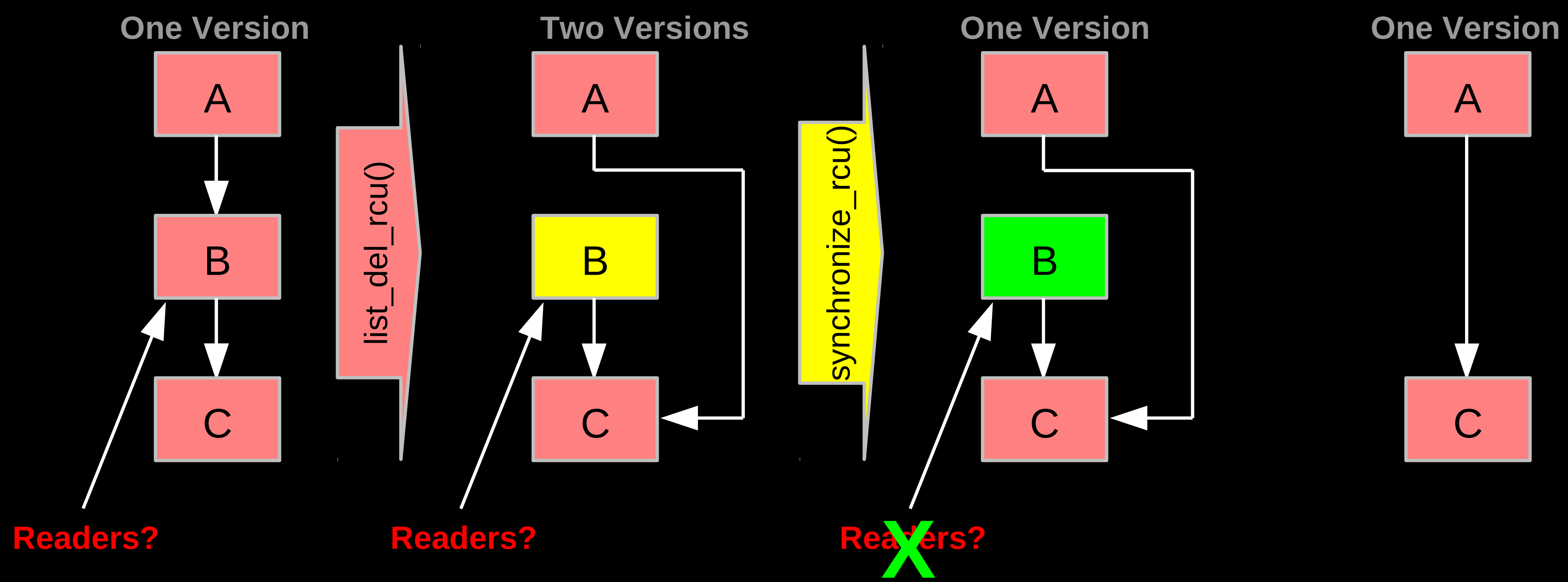
Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers



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

RCU Removal From Linked List

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



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

How Can RCU Tell When Readers Are Done???

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

Waiting for Pre-Existing Readers: QSBR

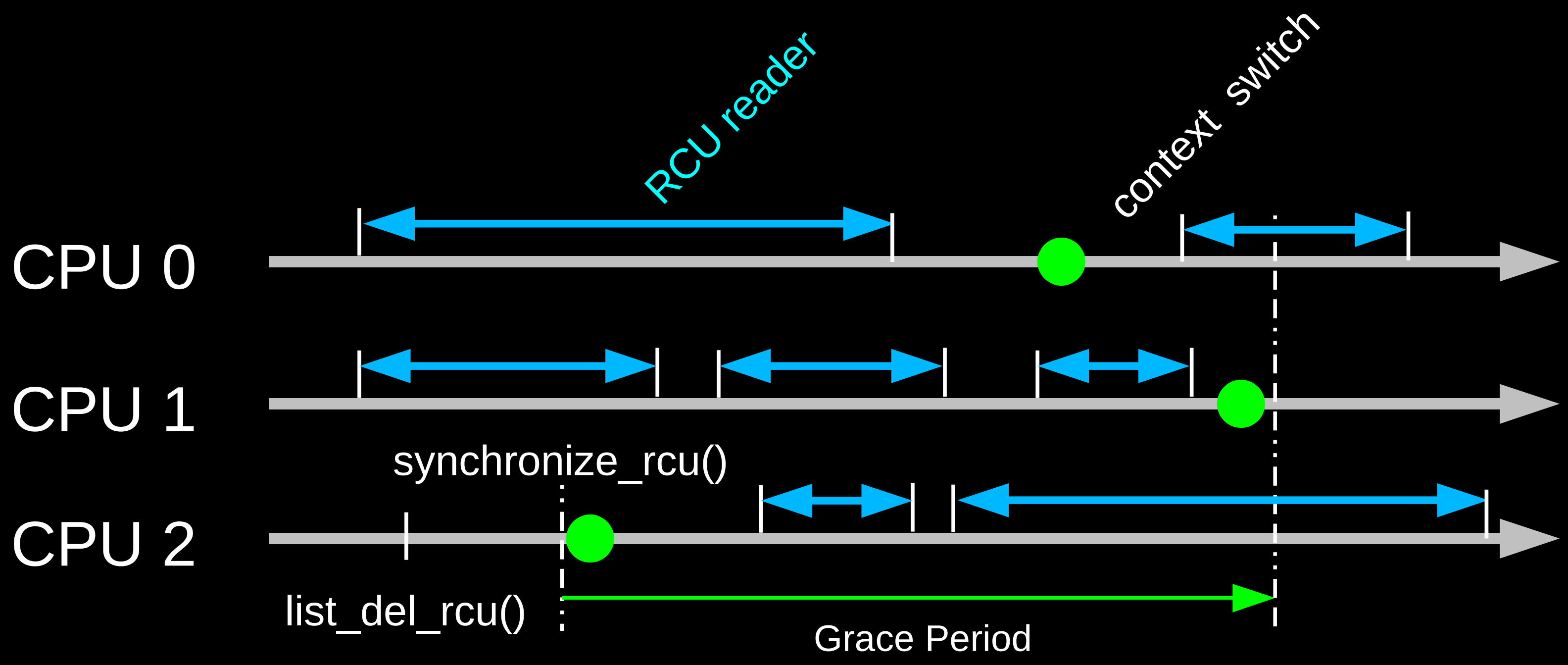
- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block
- CPU context switch means all that CPU's prior readers are done
- *Grace period* ends after all CPUs execute a context switch

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block
- CPU context switch means all that CPU's prior readers are done
- *Grace period* ends after all CPUs execute a context switch



The Unanswered Question

- But how can a primitive that doesn't affect machine state possibly be a useful synchronization primitive?

The Unanswered Question

- But how can a primitive that doesn't affect machine state possibly be a useful synchronization primitive?
  - The developer must not place synchronize_rcu() within an RCU read-side critical section
  - RCU synchronizes not via machine state, but rather the developer

The Unanswered Question

- But how can a primitive that doesn't affect machine state possibly be a useful synchronization primitive?
  - The developer must not place synchronize_rcu() within an RCU read-side critical section
  - RCU synchronizes not via machine state, but rather the developer
  - RCU achieves synchronization via social engineering!

Toy Implementation of RCU: 20 Lines of Code

- Read-side primitives:

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

- Update-side primitives

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

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

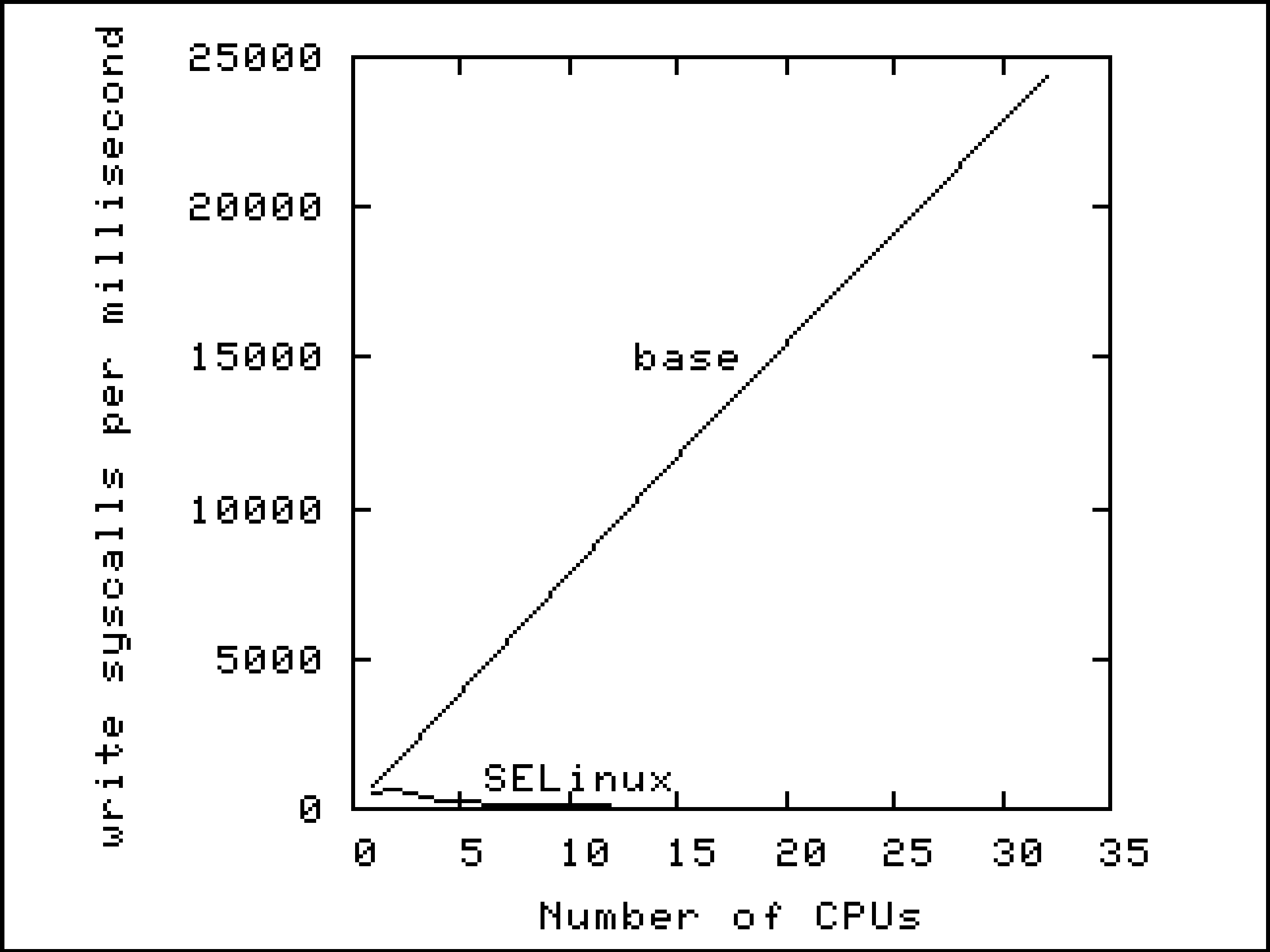
Toy Implementation of RCU on SC: 7 Lines of Code

```
void synchronize_rcu(void)
{
        int cpu;

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

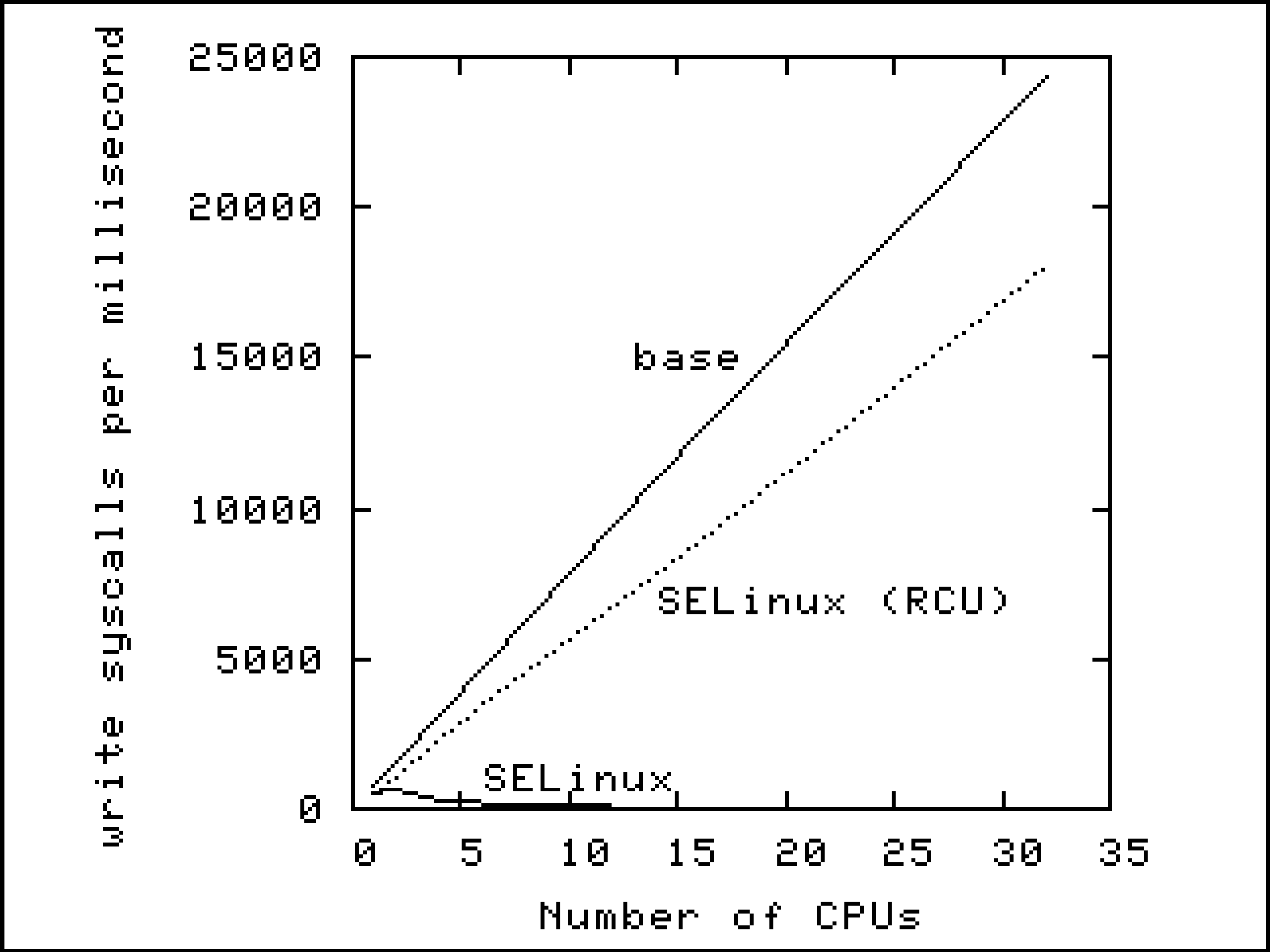
And some people still insist that RCU is complicated... ;-)

Linux Kernel write() System Call: SELinux (Logscale)



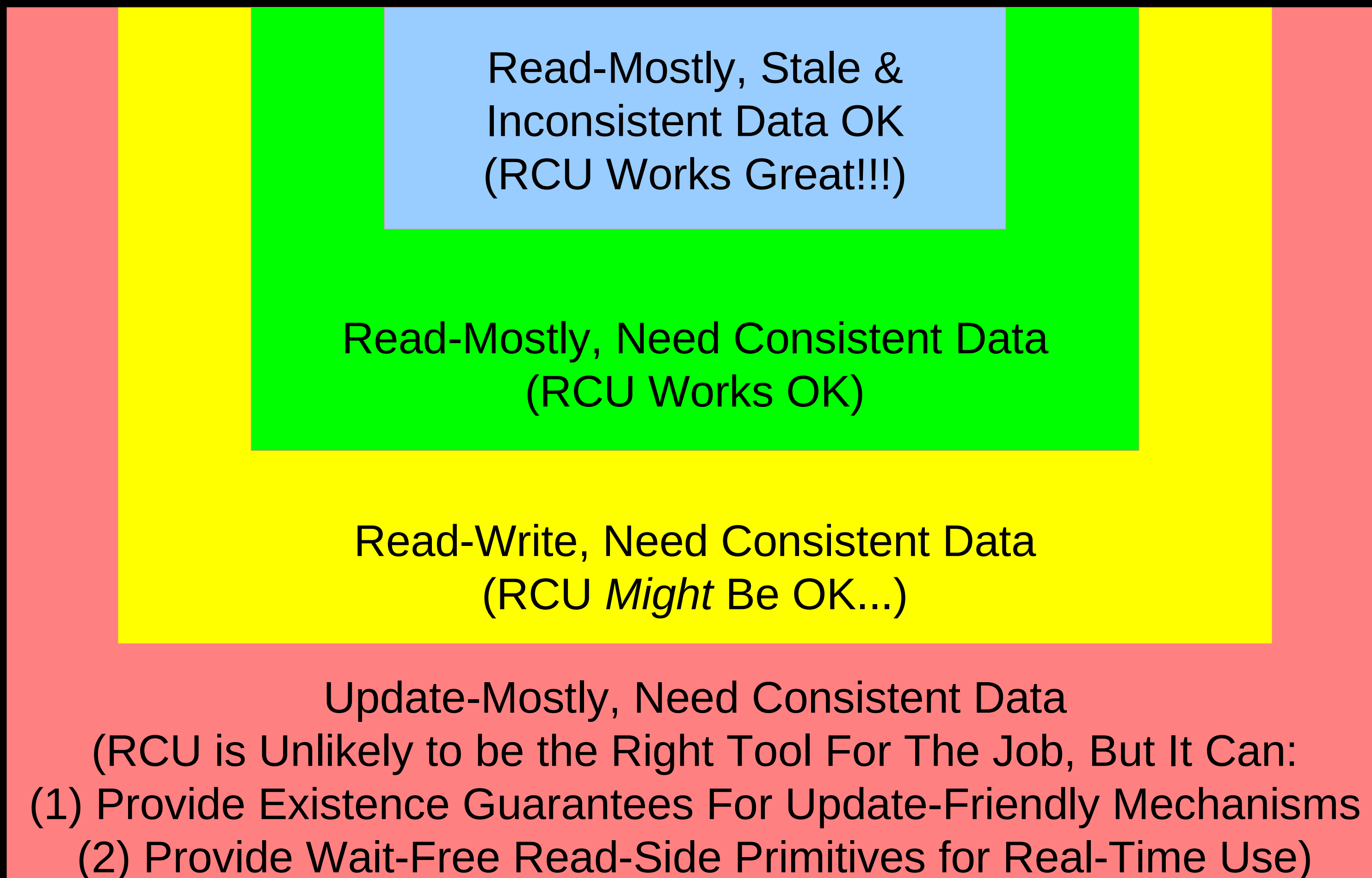
Adding CPUs makes SELinux *slower!!!*

Linux Kernel write() System Call: SELinux (RCU)

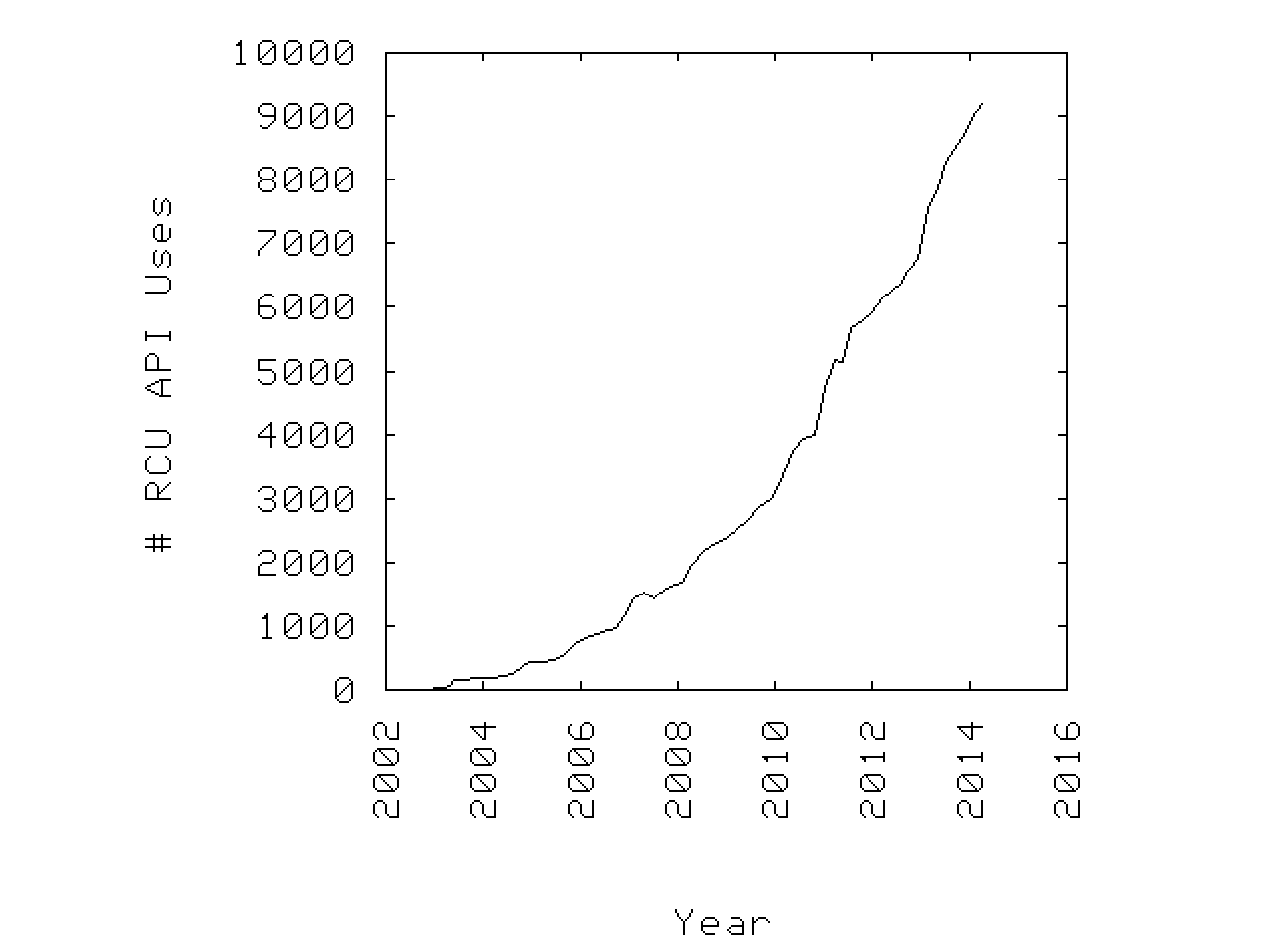


RCU provides linear scalabilty *and* order-of-magnitude improvements

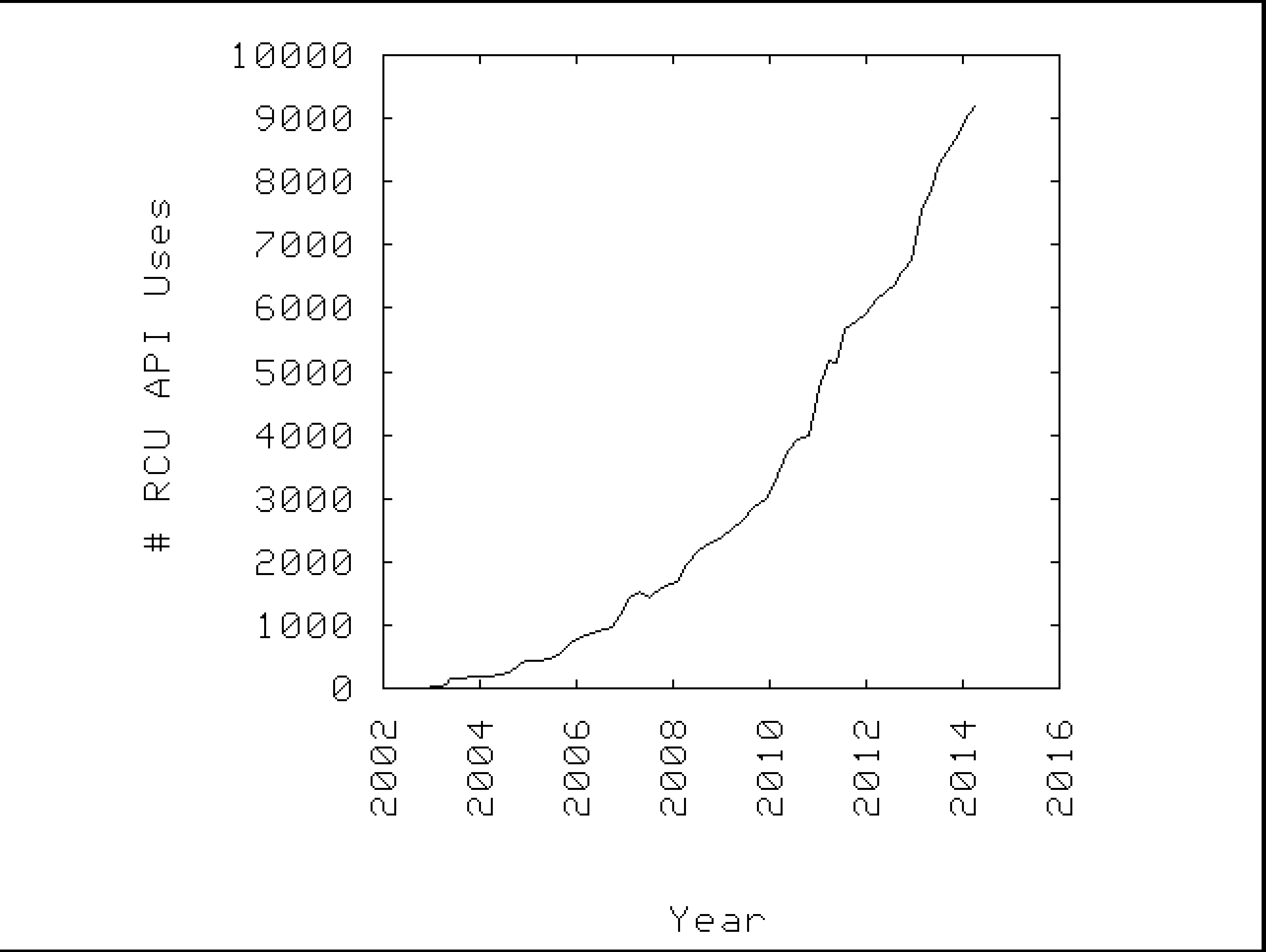
RCU Area of Applicability



RCU Applicability to the Linux Kernel



RCU Applicability to the Linux Kernel



Which is great – but how are we validating all this???

To Probe Further Into 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)

Linux Kernel Validation: A Grand Challenge

Linux Kernel Validation: A Grand Challenge

- Suppose that there is an RCU bug that occurs on average once every million years of execution time

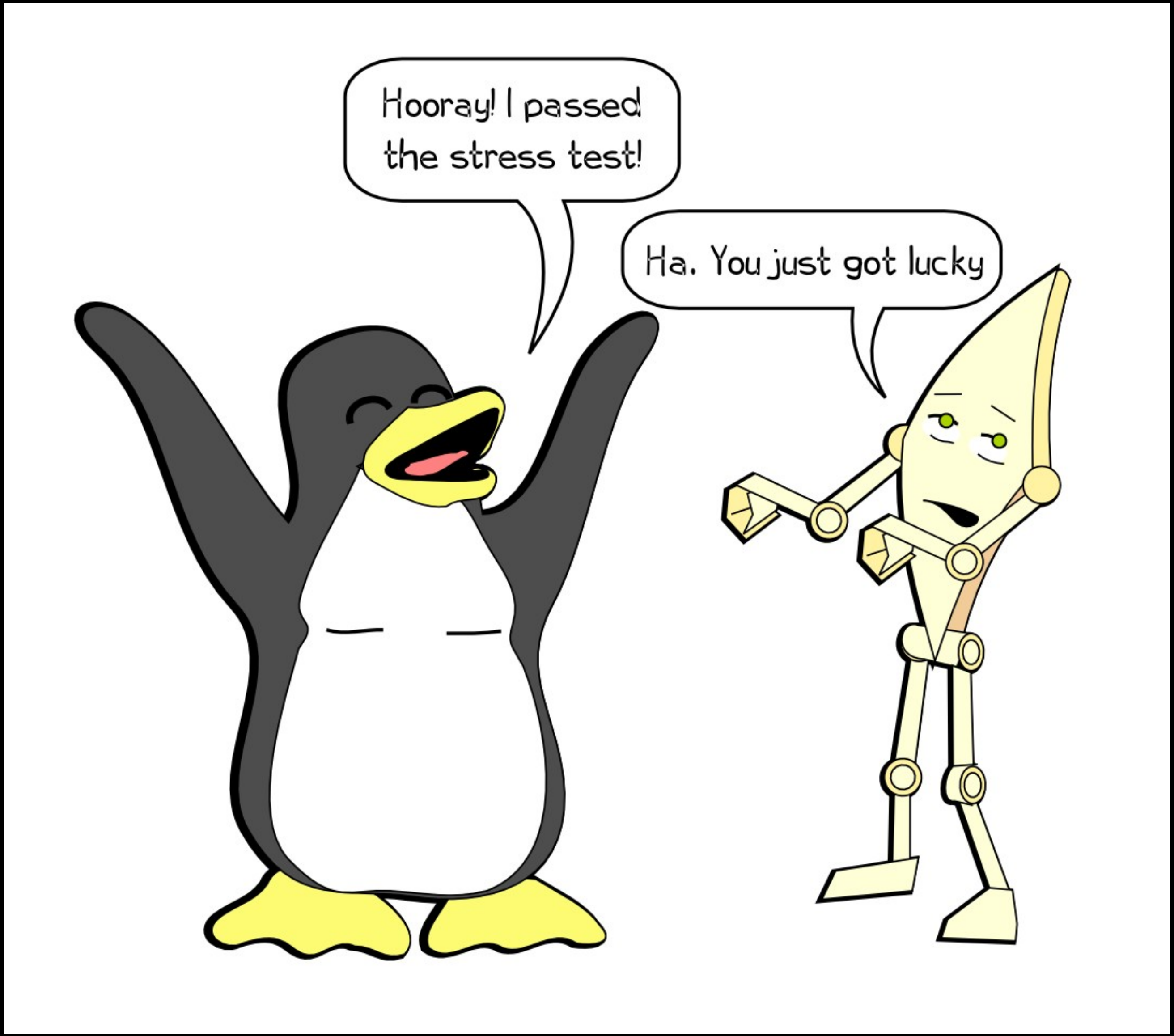
Linux Kernel Validation: A Grand Challenge

- Suppose that there is an RCU bug that occurs on average once every million years of execution time
- There are now more than one billion Linux kernel instances

Linux Kernel Validation: A Grand Challenge

- Suppose that there is an RCU bug that occurs on average once every million years of execution time
- There are now more than one billion Linux kernel instances
- Therefore this bug is exercised about three times per ***day*** across the installed base!!!

Limits to Test-Based Validation



http://paulmck.livejournal.com/36150.html

Linux Kernel Validation State of the Art & Mitigations

Linux Kernel Validation Mitigations

- Why are we getting reasonable reliability on 1G instances???
  - At >15M lines of code, there ***are*** bugs
  - Million-year bugs happen about ***three times per day***
  - And some bugs do get through

Linux Kernel Validation Mitigations

- Why are we getting reasonable reliability on 1G instances???
  - At >15M lines of code, there ***are*** bugs
  - Million-year bugs happen about ***three times per day***
  - And some bugs do get through
- The bulk of Linux's installed base has few CPUs
  - Many SMP bugs found and fixed on larger server systems
  - But the CPU counts of “small” embedded systems increasing
- The bulk of Linux's installed base has predictable workload
  - System testing can find most of the relevant bugs
  - But smartphones are becoming general-purpose systems, which will render system testing less effective
- Fortunately lots of validation: testing and tooling!!!

Linux Kernel Validation Overview

- Code review: 10,000 eyes
  - Not that review has kept pace with change rate and complexity!
  - From v3.11 to v3.12:
    - 8636 files changed, 587981 insertions(+), 264385 deletions(-)
- Unit/Stress tests
  - rcutorture, locktest, kernbench, hackbench, ...
  - Linux Test Project, Dave Jones's Trinity (quite effective lately)
- Automated/recurring testing
  - Stephen Rothwell's -next testing
  - Fengguang Wu's kbuild test robot (see next slide)
  - Frequent testing from many individuals and organizations
- Tools: sparse, lockdep, coccinelle, smatch, ...
- A big “Thank You!!!” to everyone helping with this!!!

Future Validation Needs: RCU Anecdotes

- As with airplane safety, you need to look beyond bugs in use:
  - "Near misses" caught by distro testing
    - Recent day-1 RCU CPU stall warning bug (Michal Hocko &c)
    - Shortcoming in my development methods: I need to take diagnostic code more seriously
  - "Near misses" caught by mainline testing
    - Mid-2011 v3.0-rc7 RCU/interrupt/scheduler race
    - RCU is becoming more intertwined with the rest of the kernel: I need to work to increase the isolation between RCU and the rest of the kernel
  - "Near misses" caught by my testing
    - Late 2012 day-1 RCU initialization race
    - See next slide...
- That said, in RCU "day 1" is a slippery concept
  - Three categories of statements in RCU remain from v2.6.12

Late 2012 “Day-1” RCU initialization Race

1. CPU 0 completes grace period, starts new one, cleaning up and initializing up through first leaf rcu_node structure
2. CPU 1 passes through quiescent state (new grace period!)
3. CPU 1 does rcu_read_lock() and acquires reference to A
4. CPU 16 exits dyntick-idle mode (back on ***old*** grace period)
5. CPU 16 removes A, passes it to call_rcu()
6. CPU 16 becomes associates callback with next grace period
7. CPU 0 completes cleanup/initialization of rcu_node structures
8. CPU 16 associates callback with now-current grace period
9. All remaining CPUs pass through quiescent states
10. Last CPU performs cleanup on all rcu_node structures
11. CPU 16 notices end of grace period, advances callback to “done” state
12. CPU 16 invokes callback, freeing A (too bad CPU 1 is still using it)

RCU reviewers are smart, but I cannot expect them to find this.

Linux Kernel Validation: Future Possibilities

Validation Via Model Checking

- Formal methods sometimes used by practitioners:
  - QRCU: http://lwn.net/Articles/243851/
  - dyntick-idle: http://lwn.net/Articles/279077/
  - Userspace RCU: http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159
  - NO_HZ_FULL_SYSIDLE also validated via Promela (twice!)
- However, going from C to Promela not free of pitfalls
  - Converting C to Promela on each release does not scale!
  - Verifies design, yes, but useless for regression testing
- And the need to use formal methods is often an indication that some simpler method will soon be available

Validation Via Model Checking

- Researchers' traditional focus:
  - Full validation of *all* behaviors of the system
    - Too bad a description of all behaviors can be as big as the system itself
  - Strong ordering
    - Too bad that all modern systems are weakly ordered, even x86
  - Special-purpose languages (e.g., Promela/spin)
    - Too bad that most parallel code is in general-purpose languages like C/C++
- Richard Bornat, 2011:
  - Our job is to validate the code developers write, in the environment they write it in, and in the language that they write it.
- A number of researchers have been taking this to heart
  - Peter Sewell, Susmit Sarkar, Jade Alglave, Daniel Kroening, Michael Tautschnig, Alexey Gotsman, Noam Riznetsky, Hongseok Yang, ...

Concurrency and Validation: Sewell & Sarkar's Group

- Formalization of weak-memory models (x86, Power, ARM)
  - http://lwn.net/Articles/470681/
- Tools for full state-space search of concurrent code

```
PPC IRIW.litmus
""
(* Traditional IRIW. *)
{
0:r1=1; 0:r2=x;
1:r1=1;         1:r4=y;
2:      2:r2=x; 2:r4=y;
3:      3:r2=x; 3:r4=y;
}
 P0            | P1            | P2                 | P3                 ;
 stw r1,0(r2)  | stw r1,0(r4)  | lwz r3,0(r2)       | lwz r3,0(r4)       ;
               |               | sync               | sync               ;
               |               | lwz r5,0(r4)       | lwz r5,0(r2)       ;

exists
(2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
```

Concurrency and Validation: Sewell & Sarkar's Group

- Extremely valuable tool
  - Semi-definitive answers for atomic operations and memory barriers
  - Explores every state that a real system could possibly enter
  - Near production quality
- Some shortcomings:
  - Need to translate code to assembly language
  - Does not handle arbitrary loops or arrays
  - Only handles very small code sequences
  - Applies to Power, ARM, C/C++11, but not generic Linux barriers
  - ~14 CPU-hours and ~10GB to validate example, 3.3MB of output
    - Failures detected more quickly
    - Omitting sync instructions detects failure in less than three CPU minutes
    - And knowing in 14 hours is better than just not knowing!
- Important milestone in handling real-world parallelism

Validation Via Model Checking: Alglave, Kroening, and Tautschnig

- Programming languages might be Turing complete, but you can get a long way with finite state machines: Any real system is FSM
- Finite state machines represented by logic expressions
  - Assertions can be tested with boolean satisfiabilty tester (SAT)
  - Memory model captured (partially) as additional constraints
- SAT is NP complete
  - But full state-space searches are no picnic, either
  - And much progress on SAT: million-variable problems now feasible
- Easily scripted:

```
#!/bin/sh
goto-cc -o $1.goto $1.c
goto-instrument --wmm power $1.goto $1_power.goto
nthreads=`grep __CPROVER_ASYNC_ $1.c | wc -l`
nthreads=`expr $nthreads + 1`
satabs --concurrency --full-inlining --max-threads $nthreads $1_power.goto
```

Multithreaded Model Checking: IRIW Example Input

```
int __unbuffered_cnt=0;
int __unbuffered_p0_EAX=0;
int __unbuffered_p0_EDX=0;
int __unbuffered_p1_EAX=0;
int __unbuffered_p1_EDX=0;
int x=0;
int y=0;

void * P2(void * arg) {
  x = 1;
  // Instrumentation for CPROVER
  asm("sync ");
  __unbuffered_cnt++;
}

void * P3(void * arg) {
  y = 1;
  // Instrumentation for CPROVER
  asm("sync ");
  __unbuffered_cnt++;
}
```

```
void * P0(void * arg) {
  __unbuffered_p0_EAX = x;
  asm("sync ");
  __unbuffered_p0_EDX = y;
  // Instrumentation for CPROVER
  asm("sync ");
  __unbuffered_cnt++;
}

void * P1(void * arg) {
  __unbuffered_p1_EAX = y;
  asm("sync ");
  __unbuffered_p1_EDX = x;
  // Instrumentation for CPROVER
  asm("sync ");
  __unbuffered_cnt++;
}
```

Multithreaded Model Checking: IRIW Example Input

```
int main() {
  __CPROVER_ASYNC_0: P0(0);
  __CPROVER_ASYNC_1: P1(0);
  __CPROVER_ASYNC_2: P2(0);
  __CPROVER_ASYNC_3: P3(0);
  __CPROVER_assume(__unbuffered_cnt==4);
  assert(__unbuffered_p0_EAX==0 || __unbuffered_p0_EDX == 1 ||
         __unbuffered_p1_EAX==0 || __unbuffered_p1_EDX == 1);
  return 0;
}
```

Multithreaded Model Checking: IRIW Example Output

```
. . .

Statistics of refiner:
Invalid states requiring more than 1 passive thread: 2
Spurious assignment transitions requiring more than 1 passive thread: 0
Spurious guard transitions requiring more than 1 passive thread: 0
Total transition refinements: 48
Transition refinement iterations: 10

VERIFICATION SUCCESSFUL
```

Same result as cppmem, but ***much*** faster: 2.61 CPU seconds vs ~14 CPU hours
Omitting sync instructions slows down to 134 CPU seconds: larger expressions

But They Were Not Satisfied With This...

But They Were Not Satisfied With This...

"Herding cats: Modelling, simulation, testing, and data-mining for weak memory"
Alglave, Maranget, and Tautschnig, to appear in TOPLAS.

IRIW According to the "herd" Tool

```
. . .

2:r3=1; 2:r5=1; 3:r3=1; 3:r5=0;
2:r3=1; 2:r5=1; 3:r3=1; 3:r5=1;
No
Witnesses
Positive: 0 Negative: 15
Condition exists (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
Observation IRIW Never 0 15
Hash=41423414f4e33c57cc1c9f17cd585c4d
```

Same result as cppmem and goto-cc/goto-instrument/satabs, but even faster:
16 ***milliseconds*** (vs. 2.61 CPU sec for goto... and ~14 CPU hours for ppcmem
You omitted the sync instructions? Still 16 milliseconds to validate failure!

Two orders of magnitude improvement over goto..., and ***six*** orders of magnitude
Improvement over ppcmem. So maybe the axiomatic approach is even better
use of SAT solvers! :-)

Tantalizing Possibilities

- Might I add comments to Linux-kernel RCU marking sections of code that can be formally verified?
  - Rerun the verification on each release
  - Or even as part of each testing cycle
- What is needed to make this happen?
  - Much better idea of the scope of the SAT-based and axiomatic formal verification approaches
  - Increased reliability of the formal verification software
  - Scaffolding and assertions to be automatically incorporated
    - Hopefully this can be a small matter of scripting

Summary

- Linux kernel makes heavy use of weak ordering
  - Split counters, memory allocators, RCU, …
- Linux-kernel validation grand challenge:
  - One billion instances: Million-year bugs happening three times per day!
- Substantive validation technology:
  - Per-commit build/boot/test, lock dependency checking, static analysis, stress testing, occasional use of formal verification
- Important mitigation factors:
  - Extensive testing on 4096 CPUs, real-time use, most of installed base having few CPUs, …
- But more is needed: Will I be able to add powerful formal verification methods to my RCU validation suite?

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?