Formal Verification and Linux-Kernel Concurrency
Overview

- Two Definitions and a Consequence
- Current RCU Regression Testing
- How Well Does Linux-Kernel Testing Really Work?
- Why Formal Verification?
- Formal Verification and Regression Testing: Requirements
- Formal Verification Challenge
Two Definitions and a Consequence
Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs
Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

Therefore, any non-trivial reliable software system contains at least one bug that you don't know about
Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

Therefore, any non-trivial reliable software system contains at least one bug that you don't know about

Yet there are more than a billion users of the Linux kernel
Two Definitions and a Consequence

▪ A non-trivial software system contains at least one bug
▪ A reliable software system contains no known bugs

▪ Therefore, any non-trivial reliable software system contains at least one bug that you don't know about

▪ Yet there are more than a billion users of the Linux kernel
  – In practice, validation is about reducing risk
  – Can formal verification now take a front-row seat in this risk reduction?
Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

Therefore, any non-trivial reliable software system contains at least one bug that you don't know about

Yet there are more than a billion users of the Linux kernel
- In practice, validation is about reducing risk
- Can formal verification now take a front-row seat in this risk reduction?

What would need to happen for me to include formal verification in my RCU regression testing?
Current RCU Regression Testing
Current RCU Regression Testing
But First, What Is RCU (Read-Copy Update)?
RCU Is A Synchronization Mechanism That Avoids Contention and Expensive Hardware Operations

16-CPU 2.8GHz Intel X5550 (Nehalem) System

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

Typical synchronization mechanisms do this a lot, plus suffer from contention.

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

Want to be here!
RCU Has Exceedingly Lightweight Readers

- In non-preemptible (run-to-block) environments, lightest-weight conceivable read-side primitives
  - `#define rcu_read_lock()`
  - `#define rcu_read_unlock()`
  - RCU readers are clearly extremely weakly ordered

- Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency

- Uses indirect reasoning to determine when readers are done
  - In preemptible environments, `rcu_read_lock()` and `rcu_read_unlock()` manipulate per-thread variables

- References:
  - Additional references in backup slides
Publication of And Subscription to New Data

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

See “To probe deeper” slides for more information
RCU Performance: Read-Only Hash Table

RCU and hazard pointers scale quite well!!!
RCU Area of Applicability

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)
RCU Applicability to the Linux Kernel
Current RCU Regression Testing
Current RCU Regression Testing

▪ Stress-test suite: “rcutorture”
  – http://lwn.net/Articles/154107/, http://lwn.net/Articles/622404/

▪ “Intelligent fuzz testing”: “trinity”

▪ Test suite including static analysis: “0-day test robot”
  – https://lwn.net/Articles/514278/

▪ Integration testing: “linux-next tree”
  – https://lwn.net/Articles/571980/
Current RCU Regression Testing

- Stress-test suite: “rcutorture”
  - http://lwn.net/Articles/154107/, http://lwn.net/Articles/622404/

- “Intelligent fuzz testing”: “trinity”

- Test suite including static analysis: “0-day test robot”
  - https://lwn.net/Articles/514278/

- Integration testing: “linux-next tree”
  - https://lwn.net/Articles/571980/

- Above is old technology – but not entirely ineffective
  - 2010: wait for -rc3 or -rc4.  2013: No problems with -rc1

- Formal verification in design, but not in regression testing
  - http://lwn.net/Articles/243851/, https://lwn.net/Articles/470681/, https://lwn.net/Articles/608550/
How Well Does Linux-Kernel Testing Really Work?
Example 1: RCU-Scheduler Mutual Dependency

RCU

Scheduler

Synchronization

Schedule Threads
Priority Boosting
Interrupt Handling
So, What Was The Problem?

- Found during testing of Linux kernel v3.0-rc7:
  - RCU read-side critical section is preempted for an extended period
  - RCU priority boosting is brought to bear
  - RCU read-side critical section ends, notes need for special processing
  - Interrupt invokes handler, then starts softirq processing
  - Scheduler invoked to wake ksoftirqd kernel thread:
    - Acquires runqueue lock and enters RCU read-side critical section
    - Leaves RCU read-side critical section, notes need for special processing
    - Because in_irq() returns false, special processing attempts deboosting
    - Which causes the scheduler to acquire the runqueue lock
    - Which results in self-deadlock
  - (See http://lwn.net/Articles/453002/ for more details.)

- Fix: Add separate “exiting read-side critical section” state
  - Also validated my creation of correct patches – without testing!

Note: Remains a bug even under SC
Example 1: Bug Was Located By Normal Testing
Example 2: Grace Period Cleanup/Initialization Bug

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 associates callback with next grace period
7. CPU 0 completes cleanup/initialization of rcu_node structures
8. CPU 16 callback associated 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)

Not found via Linux-kernel validation: In production for 5 years!
Example 2: Grace Period Cleanup/Initialization Bug

Note: Remains a bug even under SC
Example 2: Grace Period Cleanup/Initialization Fix

Grace Period 0
Grace Period intermission
Grace Period 1
Grace Period intermission
Grace Period 0
Grace Period intermission
Grace Period 1
Grace Period intermission
Grace Period 0
Grace Period intermission
Grace Period 1

CPU 0
CPU 1
CPU 15
CPU 16
CPU 17
CPU 31

QS
Read A
Idle
Remove A
Idle
Idle

Intermission
Intermission
Intermission
Intermission
Intermission

Cannot yet free A
Example 1 & Example 2 Results

- Example 1: Bug was located by normal Linux test procedures
- Example 2: Bug was missed by normal Linux test procedures
  - Not found via Linux-kernel validation: In production for 5 years!
  - On systems with up to 4096 CPUs...
- Both are bugs even under sequential consistency
- Can formal verification do better?
Why Formal Verification?
Why Formal Verification?

- At least one billion embedded Linux devices
  - A bug that occurs once per million years manifests three times per day
  - But assume a 1% duty cycle, 10% in the kernel, and 1% of that in RCU
  - 10,000 device-years of RCU per year: \( p(RCU) = 10^{-5} \)
Why Formal Verification?

- At least one billion embedded Linux devices
  - A bug that occurs once per million years manifests three times per day
  - But assume a 1% duty cycle, 10% in the kernel, and 1% of that in RCU
  - 10,000 device-years of RCU per year: \( p(RCU) = 10^{-5} \)

- At least 20 million Linux servers
  - A bug that occurs once per million years manifests twice per month
  - Assume 50% duty cycle, 10% in the kernel, and 1% of that in RCU
  - 10,000 system-years of RCU per year: \( p(RCU) = 5(10^{-4}) \)
Why Formal Verification?

- At least one billion embedded Linux devices
  - A bug that occurs once per million years manifests three times per day
  - But assume a 1% duty cycle, 10% in the kernel, and 1% of that in RCU
  - 10,000 device-years of RCU per year: $p(RCU) = 10^{-5}$

- At least 20 million Linux servers
  - A bug that occurs once per million years manifests twice per month
  - Assume 50% duty cycle, 10% in the kernel, and 1% of that in RCU
  - 10,000 system-years of RCU per year: $p(RCU) = 5(10^{-4})$

- But assume bugs are races between pairs of random events
  - N-CPU probability of RCU race bug: $p(bug) = (p(RCU)/N)^2 N(N-1)/2$
  - Assume rcutorture $p(RCU)=1$, compute rcutorture speedup:
    - Embedded: $10^{10}$: 36.5 days of rcutorture testing covers one year
    - Server: $4(10^6)$: 250 years of rcutorture testing covers one year
    - Linux kernel releases are only about 60 days apart: RCU is moving target
How Does RCU Work Without Formal Verification?

- What is validation strategy for 20M server systems?
  - Other failures mask those of RCU, including hardware failures
    - I know of no human artifact with a million-year MTBF
  - Increasing CPUs on test system increases race probability
    - And many systems have relatively few CPUs
  - Rare but critical operations can be forced to happen more frequently
    - CPU hotplug, expedited grace periods, RCU barrier operations...
  - Knowledge of possible race conditions allows targeted tests
    - Plus other dirty tricks learned in 25 years of testing concurrent software
  - Formal verification *is* used for some aspects of RCU design
    - Dyntick idle, sysidle, NMI interactions

- But it would be valuable to use formal verification as part of RCU's regression testing!
Formal Verification and Regression Testing: Requirements
Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required
   - Automatic discarding of irrelevant portions of the code
   - Manual translation provides opportunity for human error

(2) Correctly handle environment, including memory model
   - The QRCU validation benchmark is an excellent cautionary tale

(3) Reasonable memory and CPU overhead
   - Bugs must be located in practice as well as in theory
   - Linux-kernel RCU is 15KLoC and release cycles are short

(4) Map to source code line(s) containing the bug
   - “Something is wrong somewhere” is not a helpful diagnostic: I already know bugs exist

(5) Modest input outside of source code under test
   - Preferably glean much of the specification from the source code itself (empirical spec!)

(6) Find relevant bugs
   - Low false-positive rate, weight towards likelihood of occurrence (fixes create bugs!)
Formal Validation Tools Used and Regression Testing

- **Promela and Spin**
  - Holzmann: “The Spin Model Checker”
  - I have used Promela/Spin in design for more than 20 years, but:
    - Limited problem size, long run times, large memory consumption
    - Does not implement memory models (assumes sequential consistency)
    - Special language, difficult to translate from C

- **ARMMEM and PPCMEM (2)**
  - Alglave, Maranget, Pawan, Sarkar, Sewell, Williams, Nardelli: “PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models”
    - Very limited problem size, long run times, large memory consumption
    - Restricted pseudo-assembly language, manual translation required

- **Herd (2, 3)**
  - Alglave, Maranget, and Tautschnig: “Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”
    - Very limited problem size (but much improved run times and memory consumption)
    - Restricted pseudo-assembly language, manual translation required

Useful, but not for regression testing
Cautiously Optimistic For Future CBMC Version

(1) Either automatic translation or no translation required
   - No translation required from C, discards irrelevant code quite well

(2) Correctly handle environment, including memory model
   - SC and TSO, hopefully will do other memory models in the future

(3) Reasonable memory and CPU overhead
   - OK for Tiny RCU and some tiny uses of concurrent RCU
   - Jury is out for concurrent linked-list manipulations
   - “If you live by heuristics, you will die by heuristics”

(4) Map to source code line(s) containing the bug
   - Yes, reasonably good backtrace capability

(5) Modest input outside of source code under test
   - Yes, modest boilerplate required, can use existing assertions

(6) Find relevant bugs
   - Jury still out

Ongoing Work

- Ahmed, Groce, and Jensen: Use mutation generation and formal verification to find holes in rcutorture
- Liang, Tautschnig, and Kroening: Experiments verifying RCU and uses of RCU using CBMC
- Alglave: Derive formal memory model for Linux kernel
  - Including RCU
Formal Verification Challenge
Formal Verification Challenge

- Testing has many shortcomings
  - Cannot find bugs in code not exercised
  - Cannot reasonably exhaustively test even small software systems

- Nevertheless, a number of independently developed test harnesses have found bugs in Linux-kernel RCU
  - Trinity, 0-day test robot, -next testing

- As far as I know, no independently developed formal-verification model has yet found a bug in Linux-kernel RCU
  - Therefore, this challenge:
**Formal Verification Challenge**

- Can you verify SYSIDLE from C source?
  - Or, of course, find a bug

- This Verification Challenge 2:
  - [http://paulmck.livejournal.com/38016.html](http://paulmck.livejournal.com/38016.html)

- Mathieu Desnoyers and I verified (separately) with Promela:
  - [https://www.kernel.org/pub/linux/kernel/people/paulmck/Validation/sysidle/](https://www.kernel.org/pub/linux/kernel/people/paulmck/Validation/sysidle/)

- But neither Promela/spin is not suitable for regression testing

- Can your formal-verification tool regression-test SYSIDLE?
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?
Backup RCU Slides

**RCU Removal From Linked List**

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

---

<table>
<thead>
<tr>
<th>One Version</th>
<th>Two Versions</th>
<th>One Version</th>
<th>One Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>boa</td>
<td>boa</td>
<td>boa</td>
<td>boa</td>
</tr>
<tr>
<td>cat</td>
<td>cat</td>
<td>cat</td>
<td>cat</td>
</tr>
<tr>
<td><code>list_del_rcu()</code></td>
<td><code>synchronize_rcu()</code></td>
<td><code>kfree()</code></td>
<td><code>gnu</code></td>
</tr>
<tr>
<td><code>gnu</code></td>
<td><code>gnu</code></td>
<td><code>gnu</code></td>
<td><code>gnu</code></td>
</tr>
</tbody>
</table>

Readers? X

---
Waiting for Pre-Existing Readers

- 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
Toy Implementation of RCU: 20 Lines of Code

- **Read-side primitives:**
  
  ```c
  #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**
  
  ```c
  #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
To Probe Deeper (RCU)

- [https://queue.acm.org/detail.cfm?id=2488549](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](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](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](git://lttng.org/userspace-rcu.git) (User-space RCU git tree)
- [http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf](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](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](http://read.seas.harvard.edu/cs261/2011/rcu.html) — Harvard University class notes on RCU (Courtesy of Eddie Koher)
To Probe Deeper (1/5)

- Hash tables:

- Split counters:
  - 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
    • 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”
  - 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”
To Probe Deeper (2/5)

- **Stream-based applications:**
  - Sutton: “Concurrent Programming With The Disruptor”
    - [http://www.youtube.com/watch?v=UvE389P6Er4](http://www.youtube.com/watch?v=UvE389P6Er4)
  - Thompson: “Mechanical Sympathy”
    - [http://mechanical-sympathy.blogspot.com/](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](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/](https://lwn.net/Articles/419811/)
  - Xu: “bridge: Add core IGMP snooping support”
  - Triplett et al., “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
  - McKenney et al: “URCU-Protected Hash Tables”
    - [http://lwn.net/Articles/573431/](http://lwn.net/Articles/573431/)
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](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/](http://www.power.org/documentation/power-isa-version-2-07/)
  - Intel® 64 and IA-32 Architectures Software Developer Manuals
  - Jacobi et al: “Transactional Memory Architecture and Implementation for IBM System z”

- **Hardware lock elision: Evaluations**
  - [http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html Section 16.3](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](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](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](http://dl.acm.org/citation.cfm?id=2414731)
  - Gramoli et al: “Democratizing transactional programming”
    - [http://doi.acm.org/10.1145/2541883.2541900](http://doi.acm.org/10.1145/2541883.2541900)
To Probe Deeper (4/5)

- **RCU**
  - Desnoyers et al.: “User-Level Implementations of Read-Copy Update”
    - [http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.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”
  - McKenney: “Structured deferral: synchronization via procrastination”
    - [http://doi.acm.org/10.1145/2483852.2483867](http://doi.acm.org/10.1145/2483852.2483867)
  - McKenney et al.: “User-space RCU” [https://lwn.net/Articles/573424/](https://lwn.net/Articles/573424/)

- **Possible future additions**
  - Boyd-Wickizer: “Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels”
  - McKenney: “N4037: Non-transactional Implementation of Atomic Tree Move”
  - McKenney: “C++ Memory Model Meets High-Update-Rate Data Structures”
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”
  - McKenney, “Exploiting Deferred Destruction: An Analysis of RCU Techniques”
  - Hart, “Applying Lock-free Techniques to the Linux Kernel”
  - Olsson et al., “TRASH: A dynamic LC-trie and hash data structure”
  - Desnoyers, “Low-Impact Operating System Tracing”
  - Dalton, “The Design and Implementation of Dynamic Information Flow Tracking ...”
  - Gotsman et al., “Verifying Highly Concurrent Algorithms with Grace (extended version)”
  - 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”
Backup Promela/PPCMEM/Herd Slides
Promela Model of Incorrect Atomic Increment (1/2)

1 #define NUMPROCS 2
2
3 byte counter = 0;
4 byte progress[NUMPROCS];
5
6 proctype incremener(byte me)
7 {
8   int temp;
9
10  temp = counter;
11  counter = temp + 1;
12  progress[me] = 1;
13 }
Promela Model of Incorrect Atomic Increment (2/2)

15 init {
16   int i = 0;
17   int sum = 0;
18
19   atomic {
20     i = 0;
21     do
22       :: i < NUMPROCS ->
23         progress[i] = 0;
24         run incrementer(i);
25         i++
26       :: i >= NUMPROCS -> break
27     od;
28   }
29   atomic {
30     i = 0;
31     sum = 0;
32     do
33       :: i < NUMPROCS ->
34         sum = sum + progress[i];
35         i++
36       :: i >= NUMPROCS -> break
37     od;
38     assert(sum < NUMPROCS || counter == NUMPROCS)
39   }
40 }
PPC IRIW.litmus

(* Traditional IRIW. *)

{* 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;
}

exists

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

Fourteen CPU hours and 10 GB of memory
Herd Example Litmus Test for Incorrect IRIW

```
PPC IRIW-lwsync-f.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) 
|             |             | lwsync       | lwsync       
|             |             | lwz r5,0(r4) | lwz r5,0(r2) 

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

...

Positive: 1 Negative: 15
Condition exists (2:r3=1 \ 2:r5=0 \ 3:r3=1 \ 3:r5=0)
Observation IRIW Sometimes 1 15