Tracing and Linux-Kernel RCU
Overview

▪ Two Definitions and a Consequence
▪ Avoiding Bugs
▪ Triggering Bugs Quickly
▪ Locating Bugs Once Triggered (Tracing Is Here)
▪ Recent Improvements In Use of Tracing
▪ Possible Future Improvements (Not Just Tracing)
▪ Summary
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

It is necessary to find that bug!
Avoiding Bugs
Avoiding Bugs: Design Time

- To the extent possible, establish requirements
  - Hint: It never is completely possible!

- Understand the hardware and underlying software
  - Shameless plug: “Is Parallel Programming Hard, And, If So, What Can You Do About It?”

- Set down design (text, figures, discussion, whatever)

- In some cases, formal verification in design, for example:
  - http://lwn.net/Articles/243851/, https://lwn.net/Articles/470681/, https://lwn.net/Articles/608550/
  - But need for formal verification often means too-complex design!
Avoiding Bugs: Coding Time
Avoiding Bugs: Coding Time

- Review your code carefully (can't always count on others!)
  - Write the code long hand in pen on paper
  - Correct bugs as you go
  - Copy onto a clean sheet of paper
  - Repeat until the last two versions are identical

- What constitutes “not complex”?
  - Sequential code, and
  - You can test it line-by-line, as in a scripting language or via gdb
Avoiding Coding Bugs: Case Study

```c
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
                                      struct rcu_node *rnp)
{
    struct rcu_node *rnp_root = rcu_get_root(rsp);
    if (list_empty(rcu_task_struct(rsp, &rnp_root)))
    {
        int i;
        for (i = 0; i < 2; i++)
        {
            struct task_struct *tp;
            for (tp = list_entry(rnp_root->blocked_tasks[i]);
                 !list_empty(tp->blocked_tasks[i]);
                 tp = list_entry(tp->blocked_tasks[i]))
            {
                continue;
            }
        }
    }
    for_each_entry(tp, &rnp->tasks)
    {
        ...
    }
}
```
static void rcu_preempt_offline_tasks(struct rcu_state *rsp, struct rcu_node *rnp)
{
    int i;
    struct list_head *lp;
    struct list_head *lp_root;
    struct rcu_node *rnp_root = rcu_get_root(rsp);
    struct task_struct *tp;

    if (rnp == rnp_root)
        return;

    for (i = 0; i < 2; i++)
    {
        lp = &rnp->blocked_tasks[i];
        while (!list_empty(lp))
        {
            tp = list_entry(lp->next, struct rcu_node, rcu_node_entry);
            Spin_lock(&rnp_root->lock); /* irqs disabled */
            list_del(&tp->rcu_node_entry);
            Spin_unlock(&rnp_root->lock); /* irqs disabled */
        }
    }
}
static void rea_preact_offline_tasks(struct rea_state *rsp,
    struct rea_node *rnp)
{
    int i;
    struct list_head *lp;
    struct list_head *lp_root;
    struct rea_node *rnp_root = rea_get_root(rsp);
    struct task_struct *tp;

    if (rnp == rnp_root)
        return;
    for (i = 0; i < 25; i++)
    {
        lp = &rnp->blocked_tasks[i];
        lp_root = &rnp_root->blocked_tasks[i];
        while (!list_empty(lp))
        {
            tp = list_entry(lp->next, type_t, (*tp), rea_node.entry);
            spin_lock(&rnp_root->lock); /* iros disabled */
            list_del(&tp->rea_node.entry);
            list_add(&tp->rea_node.entry, lp_root);
            tp->rea_blocked_node = rnp_root;
            spin_unlock(&rnp_root->lock); /* iros disabled */
        }
    }
}
So, How Well Did I Do?
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
        struct rcu_node *rnp,
        struct rcu_data *rdp)
{
    int i;
    struct list_head *lp;
    struct list_head *lp_root;
    struct rcu_node *rnp_root = rcu_get_root(rsp);
    struct task_struct *tp;

    if (rnp == rnp_root) {
        WARN_ONCE(1, "Last CPU thought to be offlined?");
        return;
    }

    WARN_ON_ONCE(rnp != rdp->mynode &&
        (!list_empty(&rnp->blocked_tasks[0]) ||
         !list_empty(&rnp->blocked_tasks[1])));

    for (i = 0; i < 2; i++) {
        lp = &rnp->blocked_tasks[i];
        lp_root = &rnp_root->blocked_tasks[i];
        while (!list_empty(lp)) {
            tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
            spin_lock(&rnp_root->lock); /* irqs already disabled */
            list_del(&tp->rcu_node_entry);
            tp->rcu_blocked_node = rnp_root;
            list_add(&tp->rcu_node_entry, lp_root);
            spin_unlock(&rnp_root->lock); /* irqs remain disabled */
        }
    }
}
static int rcu_preempt_offline_tasks(struct rcu_state *rsp, 
    struct rcu_node *rnp, 
    struct rcu_data *rdp)
{
    int i;
    struct list_head *lp;
    struct list_head *lp_root;
    int retval;
    struct rcu_node *rnp_root = rcu_get_root(rsp);
    struct task_struct *tp;

    if (rnp == rnp_root) {
        WARN_ONCE(1, "Last CPU thought to be offlined?");
        return 0; /* Shouldn't happen: at least one CPU online. */
    }

    WARN_ON_ONCE(rnp != rdp->mynode &&
                  (!list_empty(&rnp->blocked_tasks[0]) ||
                   !list_empty(&rnp->blocked_tasks[1])));

    retval = rcu_preempted_readers(rnp);
    for (i = 0; i < 2; i++) {
        lp = &rnp->blocked_tasks[i];
        lp_root = &rnp_root->blocked_tasks[i];
        while (!list_empty(lp)) {
            tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
            spin_lock(&rnp_root->lock); /* irqs already disabled */
            list_del(&tp->rcu_node_entry);
            tp->rcu_blocked_node = rnp_root;
            list_add(&tp->rcu_node_entry, lp_root);
            spin_unlock(&rnp_root->lock); /* irqs remain disabled */
        }
    }

    return retval;
}
static int rcu_preempt_offline_tasks(struct rcu_state *rsp,
                      struct rcu_node *rnp,
                      struct rcu_data *rdp)
{
    int i;
    struct list_head *lp;
    struct list_head *lp_root;
    int retval;
    struct rcu_node *rnp_root = rcu_get_root(rsp);

    if (rnp == rnp_root) {
        WARN_ONCE(1, "Last CPU thought to be offlined?");
        return 0; /* Shouldn't happen: at least one CPU online. */
    }
    WARN_ON_ONCE(rnp != rdp->mynode &&
             (!list_empty(&rnp->blocked_tasks[0]) ||
             !list_empty(&rnp->blocked_tasks[1])));
    retval = rcu_preempted_readers(rnp);
    for (i = 0; i < 2; i++) {
        lp = &rnp->blocked_tasks[i];
        lp_root = &rnp_root->blocked_tasks[i];
        while (!list_empty(lp)) {
            tp = list_entry(lp->next, typeof(*tp), rcu_node_entry);
            spin_lock(&rnp_root->lock); /* irqs already disabled */
            list_del(&tp->rcu_node_entry);
            tp->rcu_blocked_node = rnp_root;
            list_add(&tp->rcu_node_entry, lp_root);
            spin_unlock(&rnp_root->lock); /* irqs remain disabled */
        }
    }
    return retval;
}
Avoiding Coding Bugs: Case Study Evaluation

- The approach worked just fine for the actual coding
- However, I should have put more effort into arriving at a simpler design
- Of course, at some point you do have to start coding
- And there is a cost to refusing to move these lists: The grace-period detection code must look at rcu_node structures whose CPUs are all offline
  - This is the right tradeoff now, but might not have been back in 2009
Avoiding Coding Bugs When Under Pressure

When you are fixing a critical bug, speed counts

The difference is level of risk
  ▶ The code is *already* broken, so less benefit to using extremely dainty process steps
  ▶ But *only* if you follow up with careful process
  ▶ Which I repeatedly learn the hard way: http://paulmck.livejournal.com/14639.html
  ▶ Failure to invest a few days in early 2009 cost me more than a month in late 2009!!!

Long-term perspective required
  ▶ And that means *you* – leave the “blame it on management” game to Dilbert cartoons
  ▶ Align with whatever management initiatives present themselves
But I Did All This And There Are Still Bugs!!!

- “Be Careful!!! It Is A Real World Out There!!!”
- The purpose of careful software-development practices is to reduce risk
- And another part of risk reduction is testing!
Triggering Bugs Quickly
Current RCU Regression Testing

- Stress-test suite: “rcutorture”
  - [Link](http://lwn.net/Articles/154107/), [Link](http://lwn.net/Articles/622404/)

- “Intelligent fuzz testing”: “trinity”
  - [Link](http://codemonkey.org.uk/projects/trinity/)

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

- Integration testing: “linux-next tree”
  - [Link](https://lwn.net/Articles/571980/)

- Test the test: Mutation testing
  - [Link](https://www.cs.cmu.edu/~agroce/ase15.pdf)

- Some reports of automated formal verification of RCU
  - For but one example, [Link](https://arxiv.org/abs/1610.03052)

- But let's look at some example bugs...
Example 1: RCU-Scheduler Mutual Dependency

- 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

All agree that grace period 1 starts after grace period 0 ends
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...
  - But as far as we know, this bug never did trigger in the field
- Both are bugs even under sequential consistency
  - Continued improvement in RCU's regression testing is clearly needed
Why Is Improvement Needed?

- A billion+ embedded Linux devices (1.4B smartphones)
  - 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: \( p(\text{RCU}) = 10^{-5} \)
Why Is Improvement Needed?

- A billion+ embedded Linux devices (1.4B smartphones)
  - 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 80 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
  - 40,000 system-years of RCU per year: \( p(RCU) = 5(10^{-4}) \)

Why Is Improvement Needed?

- A billion+ embedded Linux devices (1.4B smartphones)
  - 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 80 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
  - 40,000 system-years of RCU per year: \( p(RCU) = 5(10^{-4}) \)

- Races between RCU event pairs, \( p(bug) = p(RCU)^2 \):
  - N-CPU probability of race: 
    \[
    1 - (1 - p(bug))^N - Np(1 - p(bug))^{(N-1)}
    \]
  - Assume rcutorture \( p(RCU) = 1 \), compute rcutorture speedup:
    - Embedded: \( 10^{12} \): 7.9 days of rcutorture testing covers one year
    - Server: \( 4(10^6) \): 21.9 years of rcutorture testing covers one year
    - Linux kernel releases are only about 60 days apart: RCU is moving target

So Why Do RCU Failures Appear to be Rare?

- What is rcutorture's strategy for 80M server systems?
  - Other failures mask those of RCU, including hardware failures
    - I know of no human artifact with a million-year MTBF
    - But the Linux kernel is being used in applications that put the public at risk
  - Increasing CPUs on test system increases race probability
    - And many systems have relatively few CPUs
  - Rare but critical operations are forced to happen more frequently
    - Long-running RCU readers, CPU hotplug, expedited grace periods, RCU barrier operations, mass registration of RCU callbacks, irqs, mass return from idle, concurrent grace-period start, preemption, RCU priority boosting, quiescent-state forcing, conditional grace periods, sysidle, Tasks RCU, …
    - 16 test scenarios emphasizing different aspects of RCU
  - Knowledge of possible race conditions allows targeted tests
    - Plus other dirty tricks learned in 25 years of testing concurrent software
    - Provide harsh environment to force software to evolve quickly
Locating Bugs Once Triggered (Tracing Is Here)
Locating Bugs Once Triggered (Tracing Is Here)

- “What did I just change?” and review that code
- Break large commits into smaller commits
  - Difficulty of analyzing code grows exponentially (or worse) with size
- Look at conditions in which failure occurs, rule out bystanders
- Debug printk()s and WARN_ON()s
  - Especially if only executed after error is detected
- Pull code into userspace and use nicer debug tools
- Tracing
- Formal verification (more on this later if we have time)
The Common-Case RCU Bug is a Heisenbug!
The Common-Case RCU Bug is a Heisenbug!
But Tracing Still Sometimes Useful

- **Performance analysis of grace-period latencies**
  - Automatic grace-period-duration analysis of rcuperf runs:
    - tools/testing/selftests/rcutorture/bin/kvm-recheck-rcuperf.sh
    - tools/testing/selftests/rcutorture/bin/kvm-recheck-rcuperf-ftrace.sh

- **Non-heisenbugs in grace-period computation**
  - There are scripts, but they will not be permitted to see the light of day

- **Situations where something doesn’t happen**
  - Hard to place the printk()\/WARN\_ON() in such situations
  - Because rcutorture failed to detect an injected bug on TREE01!!!
    - But it did detect it on TREE02 through TREE08

- **Learning what RCU actually does**
  - Finding redundant execution (Frederic Weisbecker)
Tracing to Analyze Failure to Detect Injected Bug

- Injected bug: Each CPU seeing a new grace period clears other CPUs' bits, thus asserting that their quiescent states have already passed
  - Can result in too-short grace periods when other CPUs have not yet passed through a quiescent state
  - Can result in grace-period hangs by preventing up-tree propagation
    - But TREE01 has only one rcu_node structure, so no up-tree to prevent

- Note that TREE01 enables preemption, but tests RCU-bh
  - Readers need 50-ms delay to see bug, which are rare
  - Add tracing to determine when these delays are occurring

- Current result: Many ways for RCU to evade this bug!
  - Still looking for possible rcutorture improvements...
Recent Improvements In Use of Tracing
Recent Improvements In Use of Tracing

- Automatically dump ftrace buffer:
  - When rcuperf completes: Gather grace-period performance data
  - After rcutorture failures: Gather data on events leading to failure
  - When rcu_dynticks detects idle entry/exit misnesting
  - At RCU CPU stall warning time

- However, you still need to:
  - Build with CONFIG_RCU_TRACE=y
  - Enable relevant RCU trace events, preferably *before* the failure
Possible Future Improvements (Not Just Tracing)
Possible Future Improvements (Not Just Tracing)

- Run rcutorture in userspace (faster “hotplug” operations)
- Arrange to run rcutorture more often on a variety of arches
- Add more TBD nastiness to rcutorture
- More TBD self-defense checks in RCU
- Linux-kernel memory model
- Possibly formal verification in RCU regression testing...
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 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!)
   - Specifications are software and can have their own bugs

(6) Find relevant bugs
   - Low false-positive rate, weight towards likelihood of occurrence (fixes create bugs!)
Promela/spin: Design-Time Verification

- **1993: Shared-disk/network election algorithm (pre-Linux)**
  - Hadn't figured out bug injection: Way too trusting!!!
  - Single-point-of failure bug in specification: Fixed during coding
    - But fix had bug that propagated to field: Cluster partition
  - **Conclusion**: Formal verification is trickier than expected!!!

- **2007: RCU idle-detection energy-efficiency logic**
  - [http://lwn.net/Articles/243851/](http://lwn.net/Articles/243851/)
  - Verified, but much simpler approach found two years later
  - **Conclusion**: The need for formal verification is a symptom of a too-complex design

- **2012: Verify userspace RCU, emulating weak memory**
  - Two independent models (Desnoyers and myself), **bug injection**

- **2014: NMIs can nest!!! Affects energy-efficiency logic**
  - Verified Andy's code, and no simpler approach apparent thus far!!!
  - **Note**: Excellent example of **empirical specification**
PPCMEM and Herd

- Verified suspected bug in Power Linux atomic primitives
- Found bug in Power Linus spin_unlock_wait()
- Verified ordering properties of locking primitives
- Excellent memory-ordering teaching tools
  - Starting to be used more widely within IBM as a design-time tool
- PPCMEM: [http://lwn.net/Articles/470681/](http://lwn.net/Articles/470681/)
  - Accurate but slow
- Herd: [http://lwn.net/Articles/608550/](http://lwn.net/Articles/608550/)
  - Faster, but some correctness issues with RMW atomics and lwsync
  - Work in progress: Formalize Linux-kernel memory model
    - With Alglave, Maranget, Parri, and Stern, plus lots of architects
    - Hopefully will feed into improved tooling

Alglave, Maranget, Pawan, Sarkar, Sewell, Williams, Nardelli:
“PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models”
Alglave, Maranget, and Tautschnig: “Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”
C Bounded Model Checker (CBMC)

- Nascent concurrency and weak-memory functionality

- Valuable property: “Just enough specification”
  - Assertions in code act as specifications!
  - Can provide additional specifications in “verification driver” code

- Verified very small portions of RCU
  - Daniel Kroening, Oxford (publish/subscribe), myself (Tiny RCU)

- Has been used to verify substantial portion of Tree RCU
  - Lihao Liang, University of Oxford

- And of SRCU's core algorithm (plus an improved version)
  - Lance Roy, unaffiliated (improvements from Mathieu Desnoyers)

- Conclusion: Promising, especially if SAT progress continues

Not a full state-space exploration
  – Must be paired with testing?

Finds situations where scheduling decisions and memory-order changes could produce a significantly different result

More scalable than full state-space tools

Probably vulnerable to incomplete test suites
### Scorecard For Linux-Kernel C Code (Incomplete)

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Promela</th>
<th>PPCMEM</th>
<th>Herd</th>
<th>CBMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Automated</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Handle environment</td>
<td>MM</td>
<td></td>
<td>MM</td>
<td>MM</td>
</tr>
<tr>
<td>(3) Low overhead</td>
<td></td>
<td></td>
<td></td>
<td>SAT?</td>
</tr>
<tr>
<td>(4) Map to source code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) Modest input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6) Relevant bugs</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

Promela MM: Only SC: Weak memory must be implemented in model
Herd MM: Some PowerPC and ARM corner-case issues
CBMC MM: Only SC and TSO

**Note:** All four handle concurrency! (Promela has done so for 25 years!!!)
## Scorecard For Linux-Kernel C Code

<table>
<thead>
<tr>
<th></th>
<th>Promela</th>
<th>PPCMEM</th>
<th>Herd</th>
<th>CBMC</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Automated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Handle environment</td>
<td>(MM)</td>
<td>(MM)</td>
<td></td>
<td>(MM)</td>
<td></td>
</tr>
<tr>
<td>(3) Low overhead</td>
<td></td>
<td></td>
<td>SAT?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) Map to source code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) Modest input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6) Relevant bugs</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>

So why do anything other than testing?
### Scorecard For Linux-Kernel C Code

<table>
<thead>
<tr>
<th></th>
<th>Promela</th>
<th>PPCMEM</th>
<th>Herd</th>
<th>CBMC</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Automated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Handle environment</td>
<td>(MM)</td>
<td>(MM)</td>
<td>(MM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) Low overhead</td>
<td></td>
<td></td>
<td></td>
<td>SAT?</td>
<td></td>
</tr>
<tr>
<td>(4) Map to source code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(5) Modest input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6) Relevant bugs</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

So why do anything other than testing?
- Low-probability bugs can require expensive infinite testing regimen
- Large installed base will encounter low-probability bugs
- Safety-critical applications are sensitive to low-probability bugs
## Scorecard For Linux-Kernel C Code (Nidhugg TBD)

<table>
<thead>
<tr>
<th></th>
<th>Promela</th>
<th>PPCMEM</th>
<th>Herd</th>
<th>CBMC</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Automated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Green</td>
</tr>
<tr>
<td>(2) Handle environment</td>
<td>(MM)</td>
<td>(MM)</td>
<td>(MM)</td>
<td>(MM)</td>
<td>Green</td>
</tr>
<tr>
<td>(3) Low overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yellow</td>
</tr>
<tr>
<td>(4) Map to source code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Green</td>
</tr>
<tr>
<td>(5) Modest input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Green</td>
</tr>
<tr>
<td>(6) Relevant bugs</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

So why do anything other than testing?

- Low-probability bugs can require expensive infinite testing regimen
- Large installed base will encounter low-probability bugs
- Safety-critical applications are sensitive to low-probability bugs
Summary
Summary

- Tracing is a small but critically important part of RCU development tooling
- RCU is becoming more tracing-friendly over time
- Other tools making progress as well, even formal verification
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
  - “User-Level Implementations of Read-Copy Update”
- [git://lttng.org/userspace-rcu.git](git://lttng.org/userspace-rcu.git) (User-space RCU git tree)
  - Applying RCU and weighted-balance tree to Linux mmap_sem.
  - RCU-protected resizable hash tables, both in kernel and user space
  - Combining RCU and software transactional memory
- [http://wiki.cs.pdx.edu/rp/: Relativistic programming, a generalization of RCU](http://wiki.cs.pdx.edu/rp/)
- [http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/](http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/)
  - “What is RCU?” Series
  - RCU motivation, implementations, usage patterns, performance (micro+sys)
  - System-level performance for SELinux workload: >500x improvement
  - 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)
  - 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%E2%80%9C%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
  - Thompson: “Mechanical Sympathy”
    • http://mechanical-sympathy.blogspot.com/

- Read-only traversal to update location
  - Arcangeli et al: “Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel”
    • https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli_html/index.html
  - Corbet: “Dcache scalability and RCU-walk”
    • https://lwn.net/Articles/419811/
  - Xu: “bridge: Add core IGMP snooping support”
    • http://kerneltrap.com/mailarchive/linux-netdev/2010/2/26/6270589
  - Triplett et al., “Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming”
  - Howard: “A Relativistic Enhancement to Software Transactional Memory”
  - McKenney et al: “URCU-Protected Hash Tables”
    • 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

- 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
  - 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

- Hardware lock elision: Need for weak atomicity
  - Herlihy et al: “Software Transactional Memory for Dynamic-Sized Data Structures”
  - Shavit et al: “Data structures in the multicore age”
    - http://doi.acm.org/10.1145/1897852.1897873
  - Haas et al: “How FIFO is your FIFO queue?”
    - http://dl.acm.org/citation.cfm?id=2414731
  - Gramoli et al: “Democratizing transactional programming”
    - http://doi.acm.org/10.1145/2541883.2541900
To Probe Deeper (4/5)

- **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”
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?