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 software system is non-trivial if it has at least one bug
- A reliable software system has no known bugs
Two Definitions and a Consequence

- A software system is non-trivial if it has at least one bug
- A reliable software system has no known bugs

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

- A software system is non-trivial if it has at least one bug
- A reliable software system has no known bugs

Therefore, any non-trivial reliable software system has 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 software system is non-trivial if it has at least one bug
- A reliable software system has no known bugs

- Therefore, any non-trivial reliable software system has 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 software system is non-trivial if it has at least one bug
- A reliable software system has no known bugs

Therefore, any non-trivial reliable software system has 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>

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

Typical synchronization mechanisms do this a lot
The Conceptual Components of RCU

- Publishing of new data
- Subscribing to the current version of data
- Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  - Each reader continues traversing its copy of the data while a new copy might be being created concurrently by each updater
    • Hence the name read-copy update, or RCU
  - Once all pre-existing RCU readers are done with them, old versions of the data may be discarded
- In Linux kernel, frequently used to replace reader-writer locking

References:
- McKenney: “Structured Deferral: Synchronization via Procrastination”, July 2013 CACM
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 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
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 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())
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

```
synchronize_rcu()
Grace Period
```

```
remove data
Grace Period
free data
```

```
CPU 0
RCU reader
context switch
```

```
CPU 1
synchronize_rcu()
```

```
CPU 2
```
Synchronization Without Changing Machine State?

- But `rcu_read_lock()` does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

- RCU is therefore synchronization via social engineering

- As are all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”
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);
  }
  ```

Only 9 of which are needed on sequentially consistent systems
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
The Nature of Testing

- One does not simply test correctness into one's program
- Common practice applies statistical inference to test results
  - For example, “These test results show that the change reduced the program's failure rate by at least two orders of magnitude, with 99.5% confidence.”
- Bugs can of course be deterministic in nature
  - One system deterministically crashed every evening just after backups
  - But attempts to reproduce in the lab resulted in 27-hour MTBF
  - Once the bug was identified, a 12-minute MTBF test was produced
- Not perfect, but commonly used in practice
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/154107/), [http://lwn.net/Articles/622404/](http://lwn.net/Articles/622404/)

- “Intelligent fuzz testing”: “trinity”

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

- Integration testing: “linux-next tree”
  - [https://lwn.net/Articles/571980/](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/](http://lwn.net/Articles/243851/), [https://lwn.net/Articles/470681/](https://lwn.net/Articles/470681/), [https://lwn.net/Articles/608550/](https://lwn.net/Articles/608550/)
How Well Does Linux-Kernel Testing Really Work?
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 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

Not found via Linux-kernel validation: In production for 5 years!
On systems with up to 4096 CPUs...
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
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: \( p(bug) = (p(RCU)/N)^2N(N-1)/2 \)
  - 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?**

- So why can so many people use Linux-kernel RCU?
  - 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 embedded systems have very 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
Formal Verification and Regression Testing: Requirements
Formal Verification and Regression Testing: Requirements

(1) Either automatic translation or no translation required
   – Manual translation provides opportunity for human error

(2) Automatic discarding of irrelevant portions of the code
   – Manual discarding provides opportunity for human error

(3) Reasonable memory and CPU overhead
   – Bugs must be located in practice as well as in theory
     – Linux kernel is 20 million lines of code and life is short

(4) Map to source code line(s) containing the bug
   – “Something is wrong somewhere” is not a helpful diagnostic

(5) Modest input outside of source code under test
   – Preferably glean much of the specification from the source code itself
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**
  - 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 (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
Promela Model of Incorrect Atomic Increment (1/2)

```c
#define NUMPROCS 2

byte counter = 0;
byte progress[NUMPROCS];

proctype incremencer(byte me)
{
    int temp;

    temp = counter;
    counter = temp + 1;
    progress[me] = 1;
}
```
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
30   atomic {
31     i = 0;
32     sum = 0;
33     do
34       :: i < NUMPROCS ->
35         sum = sum + progress[i];
36         i++
37       :: i >= NUMPROCS -> break
38     od;
39     assert(sum < NUMPROCS || counter == NUMPROCS)
40   }

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)

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
Cautiously Optimistic For Future CBMC Version

(1) Either automatic translation or no translation required
   • No translation required from C

(2) Automatic discarding of irrelevant portions of the code
   • Seems to do this quite well (sometimes too well)

(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

(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

Ongoing Work

- Ahmed, Groce, and Jensen: Use mutation generation and formal verification to find holes in rcutorture
- Tautschnig and Kroening: Experiments verifying RCU and uses of RCU using CBMC
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

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

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

- This Verification Challenge 2:

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

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