Advances in Validation of Concurrent Software
Overview

- Validation Trends Over Time
- Current Linux Kernel Validation Directions
- Future Validation Needs
- Validation Via Model Checking
- Multithreaded Model Checking
Validation Trends Over Time
Validation Trends Over Time

- Range of validation needed
  - One-off hacked-up scripts have always been with us
    - Fix it if it fails, many bugs will go unnoticed and unexercised
  - As have systems requiring extreme validation
    - Mission-critical business applications
      - Lose lots of money if it fails
    - High-volume consumer applications
      - Low-probability failures have a high probability of occurring
      - Another way to lose lots of money if it fails
    - Autonomous space-exploration systems
      - No way to fix it
    - Safety-critical embedded systems
      - Lose lives if it fails
Validation Trends Over Time: Paul's Journey

- **1975-6**: Computer-dating program: < 5 users (data entry)
- **1977-1980**: University housing system: 2 users
- **1981-1985**: Building control system: ~100 users
  - Plus other embedded projects with similar user base
- **1986-1987**: System administrator: ~50 users
- **1988-1990**: Research prototypes: 1 user

Informal testing sufficed
Validation Trends Over Time: Paul's Journey

- **1975-6**: Computer-dating program: < 5 users (data entry)
- **1977-1980**: University housing system: 2 users
- **1981-1985**: Building control system: ~100 users
  - Plus other embedded projects with similar user base
- **1986-1987**: System administrator: ~50 users
- **1988-1990**: Research prototypes: 1 user
- **1990-2000**: Sequent DYNIX/ptx: ~6,000 sites, mission critical

Formal unit and stress testing required: “tlbtest” rather than “rcutorture”, but...
Validation Trends Over Time: Paul's Journey

- **1975-6**: Computer-dating program: < 5 users (data entry)
- **1977-1980**: University housing system: 2 users
- **1981-1985**: Building control system: ~100 users
  - Plus other embedded projects with similar user base
- **1986-1987**: System administrator: ~50 users
- **1988-1990**: Research prototypes: 1 user
- **1990-2000**: Sequent DYNIX/ptx: ~6,000 sites, mission critical
- **2001-present**: Linux kernel: ~1M – ~1G OS instances

What do we do now?
Validation: Paul's Philosophy

- Torture your code to the best of your ability, because otherwise it will torture you to the best of its ability!
Validation: Paul's Philosophy
Validation: Paul's Philosophy: Limits to Validity

Hooray! I passed the stress test!

Ha. You just got lucky.
Validation: Paul's Philosophy

- Torture your code to the best of your ability, because otherwise it will torture you to the best of its ability!
- But with a billion running instances out there, it is really hard to torture your code more viciously than the real world is going to torture it.
Validation: Paul's Philosophy

- Torture your code to the best of your ability, because otherwise it will torture you to the best of its ability!
- But with a billion running instances out there, it is really hard to torture your code more viciously than the real world is going to torture it
- And failing to torture your code more than the real world is going to torture it will result in bugs escaping into the wild
Validation: Paul's Philosophy

- Torture your code to the best of your ability, because otherwise it will torture you to the best of its ability!
- But with a billion running instances out there, it is really hard to torture your code more viciously than the real world is going to torture it
- And failing to torture your code more than the real world is going to torture it will result in bugs escaping into the wild
- Some of which will result in security exploits
Validation: Paul's Philosophy

- Torture your code to the best of your ability, because otherwise it will torture you to the best of its ability!
- But with a billion running instances out there, it is really hard to torture your code more viciously than the real world is going to torture it
- And failing to torture your code more than the real world is going to torture it will result in bugs escaping into the wild
- Some of which will result in security exploits
- On the other hand, the Linux community has been doing some really cool validation work!
Current Linux Kernel Validation Directions
Current Linux Kernel Validation Directions

- Why are we getting reasonable reliability on 1G instances???
  - At >10M lines of code, there are bugs
  - Million-year bugs happen about three times per day
  - And some bugs do get through
Current Linux Kernel Validation Directions

- Why are we getting reasonable reliability on 1G instances???
  - At >10M 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

- Lots of validation testing and tooling!!!
Linux Plumbers Conference, September 18, 2013

Linux Kernel Validation Overview

- Code review: 10,000 eyes
  - Not that review has kept pace with change rate and complexity
  - From v3.10 to v3.11:
    - 9693 files changed, 789124 insertions(+), 341338 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!!!
Fengguang Wu's kbuild test robot

tree:  git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/linux-rcu.git  rcu/dev
head:  7f797be6ab3cfe47e34ffe44a1a8ee8d6728893a
commit:  7f797be6ab3cfe47e34ffe44a1a8ee8d6728893a  [42/42]  rcu:
Consistent rcu_is_watching() naming
config: x86_64-randconfig-a0-0914 (attached as .config)

All error/warnings:

In file included from include/linux/srcu.h:33:0, 
  from include/linux/notifier.h:15, 
  from include/linux/memory_hotplug.h:6, 
  from include/linux/mmzone.h:797, 
  from include/linux/gfp.h:4, 
  from include/linux/slab.h:12, 
  from include/linux/crypto.h:24, 
  from arch/x86/kernel/asm-offsets.c:8:
include/linux/rcupdate.h: In function 'rcu_read_lock_held':
  include/linux/rcupdate.h:354:2: error: implicit declaration of function 'rcu_is_watching' [-Werror=implicit-function-declaration]
Fengguang Wu's kbuild test robot

vim +/rcu_is_watching +354 include/linux/rcupdate.h

348    * offline from RCU perspective, so check for those as well.
349    *
350 static inline int rcu_read_lock_held(void)
351 {
352     if (!debug_lockdep_rcu_enabled())
353         return 1;
354     if (!rcu_is_watching())
355         return 0;
356     if (!rcu_lockdep_current_cpu_online())
357         return 0;
Future Validation Needs
Future Validation Needs

- CPU counts will continue increasing for some time
  - Including for the low-end embedded systems that make up the bulk of the Linux kernel's installed base

- Scalability needs will force more aggressive parallelism
  - lockdep can't help much with atomic operations and memory barriers!
  - Manual inspection does not scale with Linux's rate of development
  - Additional automated inspection will be needed

- Many other needs, including validation against standards
  - To say nothing of validation of standards...

- But this presentation will focus on concurrency
Future Validation Needs: RCU Anecdotes

- As with airplane safety, you need to look beyond bugs in use:
  - 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
  - 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
  - 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 types 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.
Validation Via Model Checking
Validation Via Model Checking

- Researchers' traditional focus:
  - Strong ordering (e.g., Promela/spin)
    - Too bad that all modern systems are weakly ordered, even x86
  - Special-purpose languages
    - 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 where they write it in the language that they write it."

- A number 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)
- 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
  - Definitive answers for atomic operations and memory barriers
  - 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

- 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 a finite state machine

- Finite state machines represented by logic expressions
  - Assertions can be tested with boolean satisfiability tester (SAT)

- SAT is NP complete
  - But full state-space searches are no picnic, either
  - And much progress on SAT: million-variable problems now feasible
Code To Logic Expression

CPU 0
x = 1;
x = 2;

CPU 1
r1 = x;

- Initial value of x is zero
- Assume cache coherence (stores of 1 and 2 are ordered)
- Introduce three auxiliary variables:
  - Ls1s2: Load happened before store of 1
  - s1Ls2: Load happened between store of 1 and store of 2
  - s1s2L: Load happened after store of 2
- Expression:
  - Ls1s2 → r1 == 0 && s1Ls2 → r1 == 1 && s1s2L → r1 == 2
- Convert implication to boolean operators:
  - (!Ls1s2 || r1 == 0) && (!s1Ls2 || r1 == 1) && (!s1s2L || r1 == 2)
Initial logic expression:
\[ \neg (Ls1s2 \lor r1==0) \land \neg (s1Ls2 \lor r1==1) \land \neg (s1s2L \lor r1==2) \]

Problem: What if all three of $Ls1s2$, $s1Ls2$, $s1s2L$ are set?
- This would mean that CPU 1's load is both before and after both stores!
- Need some way to rule this out
- \[ (Ls1s2 \land \neg s1Ls2 \land \neg s1s2L) \lor (\neg Ls1s2 \land s1Ls2 \land \neg s1s2L) \lor (\neg Ls1s2 \land \neg s1Ls2 \land s1s2L) \]

Combining these:
- \[ ((Ls1s2 \land \neg s1Ls2 \land \neg s1s2L) \lor (\neg Ls1s2 \land s1Ls2 \land \neg s1s2L) \lor (\neg Ls1s2 \land \neg s1Ls2 \land s1s2L)) \land \neg (Ls1s2 \lor r1==0) \land \neg (s1Ls2 \lor r1==1) \land \neg (s1s2L \lor r1==2) \]
Code To Logic Expression

CPU 0
x = 1;
x = 2;

CPU 1
r1 = x;

- Initial logic expression:
  - (!Ls1s2 || r1==0) && (!s1Ls2 || r1==1) && (!s1s2L || r1==2)

- Problem: What if all three of Ls1s2, s1Ls2, s1s2L are set?
  - This would mean that CPU 1's load is both before and after both stores!
  - Need some way to rule this out
  - (Ls1s2 && !s1Ls2 && !s1s2L) || (!Ls1s2 && s1Ls2 && !s1s2L) || (!Ls1s2 && !s1Ls2 && s1s2L)

- Combining these:
  - ((Ls1s2 && !s1Ls2 && !s1s2L) || (!Ls1s2 && s1Ls2 && !s1s2L) || (!Ls1s2 && !s1Ls2 && s1s2L)) && (!Ls1s2 || r1==0) && (!s1Ls2 || r1==1) && (!s1s2L || r1==2)

- And this is supposed to make things simpler???
“Full” logic expression:
- \((-((\text{Ls1s2} \&\& \neg \text{s1Ls2} \&\& \neg \text{s1s2L}) \lor ((\neg \text{Ls1s2} \&\& \text{s1Ls2} \&\& \neg \text{s1s2L})) \&\& (\neg \text{Ls1s2} || r1==0) \&\& (\neg \text{s1Ls2} || r1==1) \&\& (\neg \text{s1s2L} || r1==2))\)\)
- In real life, need binary expansion of \(r1\)
- And expressions to relate the values of \(x\) to each other

There is a lot of software to analyze such expressions
- And to simplify and manipulate them
- And to generate them automatically from C code
- Which is a good thing because doing it by hand would be a pain!

In particular, there is a lot of code to determine what combinations of variables satisfies a given logic expression
C Bounded Model Checker (cbmc)

- Takes smallish C programs as input
- Generates corresponding logic expressions
- Optionally takes limits on loop unrolling
  - Arbitrary loops are not handled
  - Something about them generating logic expressions of infinite size
- Evaluates array bounds and assertions, among other things
  - This presentation will focus on assertions
  - Big benefit: Developer specifies correctness criteria
- Does not handle multithreading
  - But you have to start somewhere...
Example #1 cbmc Verification: Input

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    if (argc < 2) {
        printf("Usage: %s n
\n", argv[0]);
        return 1;
    }
    i = atoi(argv[1]);
i = i * 2 + 1;
assert(i & 0x1);
return 0;
}
```
Example #1 cbmc Verification: Output

$ cbmc even.c
file even.c: Parsing
Converting
Type-checking even
file even.c line 11 function main: function `c::atoi' is not declared
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 29 assignments
simple slicing removed 3 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 without simplifier
1476 variables, 4036 clauses
empty clause: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 0.017s
VERIFICATION SUCCESSFUL
Example #2 cbmc Verification: Input

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i;
  
  if (argc < 2) {
    printf("Usage: %s n
\n", argv[0]);
    return 1;
  }
  i = atoi(argv[1]);
  i = i * 2;
  assert(i & 0x1);
  return 0;
}
```
Example #2 cbmc Verification: Output

$ cbmc even-bad.c
...
State 22 file even-bad.c line 12 function main thread 0

```
main::1::i=2 (000000000000000000000000000000010)
```

Violated property:
```
file even-bad.c line 13 function main assertion
(_Bool)(i & 1)
```

VERIFICATION FAILED
#include <stdio.h>

extern int nondet_int(void);

int main(int argc, char *argv[])
{
    int a, b, c;

    a = nondet_int();
    b = nondet_int();
    c = nondet_int();
    if (a <= 0 || a > 1023 || b <= 0 || b > 1023 || c <= 0 || c > 1023) {
        printf("Usage: %s a b c\n", argv[0]);
        printf("Value must be 0 < v <= 1023\n", argv[0]);
        return 2;
    }
    assert(a * a * a + b * b * b != c * c * c);
    return 0;
}
Example #3 cbmc Verification: Output

```
$ cbmc fermat.c
file fermat.c: Parsing
Converting
Type-checking fermat
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 37 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 without simplifier
24573 variables, 29508 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 158.163s
VERIFICATION SUCCESSFUL
```

Why so slow?

Exhaustive testing can be faster, but often more work
Example #3 cbmc Verification: Output

$ cbmc fermat.c
file fermat.c: Parsing
Converting
Type-checking fermat
Generating GOTO Program
Adding CPROVER library
Function Pointer Removal
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 37 assignments
simple slicing removed 1 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
Running propositional reduction
Solving with MiniSAT2 without simplifier
24573 variables, 29508 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 158.163s
VERIFICATION SUCCESSFUL

Why so slow? Multiplication!!!

Exhaustive testing can be faster, but often more work
C Bounded Model Checker (cbmc) Summary

- CMU research project
- Readily available open source: http://www.cprover.org/cbmc/
- Part of several Linux distros
- Handles C code
- Reasonably robust and documented
- Does not handle general loops, but allows bounded unrolling
  - And checks to see if unrolling was sufficient
- Does not handle threading
  - Though some extensions have been prototyped
Multithreaded Model Checking
Multithreaded Model Checking

- Alglave, Kroener, and Tautschnig produced prototype system with goto-cc, goto-instrument, and satabs
- Memory model captured as additional constraints
- Easily scripted:

  ```bash
  #!/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

```c
# Multithreaded Model Checking Example: IRIW

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 * 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++;
}

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++;
}
```
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
goto-cc/goto-instrument/satubs Summary

- Oxford research project
- Readily available open source: http://www.cprover.org/wmm/
- Download source and/or x86 binaries
- Handles C code

Early days: Robustness and documentation lacking
  - Number of threads specified in four different places, no diagnostics!
  - Working versions as follows:
    - `$ sum goto-cc goto-instrument satubs`
    - `19375  4429  goto-cc`
    - `54447  5705  goto-instrument`
    - `24956  5969  satubs`

- Does not handle general loops, but allows bounded unrolling
  - And checks to see if unrolling was sufficient
Summary
Summary

- Validation of the Linux kernel increasingly challenging
  - More code to validate
  - More instances to exercise obscure bugs
  - More CPUs, memory, and other invitations to rare bugs

- Linux kernel community has risen to the challenge
  - Review, aggressive testing, tooling

- Future requirements likely to be more severe
  - Full state-space modeling might be one way forward for concurrency
  - cppmem: slow and low-level but accurate and trustworthy
  - goto-cc/goto-instrument/satabs: fast and high-level, but early days
    - Will likely be able to handle larger problems
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?