PARLab Parallel Boot Camp

Testing and Debugging Parallel Programs

Jacob Burnim
Electrical Engineering and Computer Sciences
University of California, Berkeley
Parallel Correctness Challenges

- Parallel programming presents a number of new challenges to writing correct software.
  - New kinds of bugs: data races, deadlocks, etc.
  - More difficult to test programs and find bugs.
  - More difficult to reproduce errors.

- Key Difficulty: Potential non-determinism.
  - Order in which threads execute can change from run to run.
  - Some runs are correct while others hit bugs.
• For **sequential** programs, we typically expect that same input $\Rightarrow$ same output:

$$\begin{align*}
x &= 0.7 \\
y &= 0.3 \\
\ldots \\
y &= 5.0
\end{align*}$$

Program P
But for parallel programs, threads can be scheduled differently each run:

\[
x = 0.7 \\
y = 0.3 \\
\vdots \\
y = 5.0
\]
• But for **parallel** programs, threads can be scheduled differently each run:

\[ x=0.7 \]
\[ y=0.3 \]
\[ \ldots \]
\[ y=5.0 \]
But for parallel programs, threads can be scheduled differently each run.

A bug may occur under only rare schedules.
- In 1 run in 1000 or 10,000 or ...

May occur only under some configurations:
- Particular OS scheduler.
- When machine is under heavy load.
- Only when debugging/logging is turned off!
• For **sequential** programs:
  - Create several test inputs with known answers.
  - Run the code on each test input.
  - If all tests give correct input, have some confidence in the program.
  - Have intuition about which “edge cases” to test.

• But for **parallel** programs:
  - Each run tests only a single schedule.
  - How can we test many different schedules?
  - How confident can we be when our tests pass?
 Outline

• Challenges for parallel testing.

• Random testing of parallel programs.

• Detecting and predicting parallel bugs.

• Active Random Testing of parallel programs.

• Conclusions.
Testing Parallel Programs

• **Possible Idea:** Can we just run each test thousands of times?

• **Problem:** Often not much randomness in OS scheduling.
  
  - May waste much effort, but test few different schedules.
  - **Recall:** Some schedules tend to occur only under certain configurations – hardware, OS, etc.
  
  - One easy parameter to change: load on machine.
Stress Testing

• **Idea:** Test parallel program while oversubscribing the machine.
  - On a 4-core system, run with 8 or 16 threads.
  - Run several instances of the program at a time.
  - Increase size to overflow cache/memory.
  - **Effect:** Timing of threads will change, giving different thread schedules.

• **Pro:** Very simple idea, easy to implement.
  - And often works!
• **Idea:** Run with random thread schedules.
  - E.g., insert code like:
    ```c
    if (rand() < 0.01) usleep(100);
    if (rand() < 0.01) yield();
    ```
  - Can add to only “suspicious” or “tricky” code.
  - Or use tool to seize control of thread scheduling.

• **Pros:** Still fairly simple and often effective.
  - Explores different schedules than stress testing.
  - Many tools can perform this automatically.
IBM’s ConTest: Noise-making for Java.
- Clever heuristics about where to insert delays.

Berkeley’s Thrille (C + pthreads) and CalFuzzer (Java) do simple random scheduling.
- Extensible: Write testing scheduler for your app.

Microsoft Research’s Cuzz (for .NET).
- New random scheduling algorithm with probabilistic guarantees for finding bugs.
- Available soon.

Many of these tools provide **replay** - same random number seed ==> same schedule.
Limitations of Random Scheduling

- Parallel programs have **huge** number of schedules - **exponential** in length of a run.

Explored by repeated execution.

Explored by some stress test.

Possible thread schedules.
Limitations of Random Scheduling

- Parallel programs have a huge number of schedules — exponential in length of a run.

Vast majority of schedules will never be tested.
Limitations of Random Scheduling

- Parallel programs have huge number of schedules - exponential in length of a run.

Vast majority of schedules will never be tested.

Can we find parallel errors without explicitly testing a schedule in which the error occurs?
• Challenges for parallel testing.

• Random testing of parallel programs.

• Detecting and predicting parallel bugs.

• Active Random Testing of parallel programs.

• Conclusions.
Detecting/Predicting Parallel Bugs

• Say we observe a test run of a parallel program that doesn’t obviously fail.

• Key Question: Can we find possible parallel bugs by examining the execution?
Detecting/Predicting Parallel Bugs

• Say we observe a test run of a parallel program that doesn’t obviously fail.

• **Key Question:** Can we find possible parallel bugs by examining the execution?

---

Program —> Run. —> Trace

**Race Detector:** Did a race occur in this execution?

**Race Predictor:** Could a race occur in a similar execution?
Detecting/Predicting Parallel Bugs

- Techniques/tools exist for:
  - Data races.
  - Atomicity violations.
  - Deadlocks.
  - Memory consistency errors.
Data Race Detection/Prediction

Recall: A **data race** occurs when two threads **concurrently** access the same memory, and at least one is a write.

```c
int x = 0;

Thread 1:
t1 = x;
x = t1 + 1;

Thread 2:
t2 = x;
x = t2 + 1;
```

**Data race** between two writes causes lost update - `x` can incorrectly be 1 instead of 2.
- 20+ years of research on race detection.
- Happens-Before Race Detection [Schonberg ‘89]:
  - Do two accesses to a variable occur, at least one a write, with no intervening synchronization?
  - No false warnings.
- Lockset Race Prediction [Savage, et al., ‘97]:
  - Does every access to a variable hold a common lock?
  - Efficient, but many false warnings.
- Hybrid Race Prediction [O’Callahan, Choi, 03]:
  - Combines Lockset with Happens-Before for better performance and fewer false warnings vs. Lockset.
Coverage vs. False Warnings

- **False Warning**: Tool reports a data race, but the race cannot happen in a real run.
- **Coverage**: How many of the real data races does a tool report?

- Hybrid race prediction:
  - Better coverage but more false warnings.
- Happens-Before race detection:
  - Fewer false warnings (still some, in practice) and less coverage.
Thread 1:

\[ x = 1; \]

\[ \text{lock}(L); \]

\[ y = 1; \]

\[ \text{unlock}(L); \]

Write(x) and Read(x) do not hold a common lock, so Lockset/Hybrid predicts a data race.

Thread 2:

Write(x) happens-before Read(x), so H-B detector reports no race.

\[ \text{lock}(L); \]

\[ y = 2; \]

\[ \text{unlock}(L); \]

\[ \text{if } (x == 0) \text{ ERROR} \]
Thread 1:

\[
\begin{align*}
    &x = 1; \\
    &\text{lock}(L); \\
    &y = 1; \\
    &\text{unlock}(L);
\end{align*}
\]

Thread 2:

\[
\begin{align*}
    &\text{lock}(L); \\
    &\text{if } (y == 1) \\
    &\text{if } (x == 0) \text{ ERROR} \\
    &\text{unlock}(L);
\end{align*}
\]

Write(x) and Read(x) do not hold a common lock, so Lockset/Hybrid predicts a data race.

False warning!
• Intel Thread Checker for C + pthreads.
  - Happens-Before race detection.

• Valgrind-based tools for C + pthreads.
  - Helgrind and DRD (Happens-Before).
  - ThreadSanitizer (Hybrid).

• CHESS performs race detection for .NET

• CalFuzzer and Thrille: hybrid race detection for Java and C + pthreads.
Dynamic detection and prediction tools exist for atomicity bugs, too.

```c
int balance = 0;
lock L;

@atomic
void deposit(int a) {
    lock(L);
    int t = balance;
    unlock(L);
    lock(L);
    balance = t + a;
    unlock(L);
}
```

- Parallel calls to deposit intended to happen all-at-once (atomically).
- No data races because of lock L.
- But deposit can be wrongly interrupted.
Atomicity Detection/Prediction

- CalFuzzer predicts atomicity bugs for Java. (Not yet implemented in Thrille.)
  - User must specify which methods or other blocks of code are intended to be atomic.
  - Or CalFuzzer can guess - e.g. synchronized methods, bodies of parallel loops, etc.

- Large body of research on detecting/predicting atomicity violations, but few publicly available tools.
CalFuzzer also predicts **deadlocks** for Java. (Not yet implemented in Thrille.)

```
lock L1, L2;

Thread 1:
...
lock(L1);
...
lock(L2);

Thread 2:
...
lock(L2);
...
lock(L1);
```
Aside: Static Analysis

• Have only discussed dynamic analyses.
  - Examine a real run/trace of a program.

• Static analyses predict data races, deadlocks, etc., without running a program.
  - Only examine the source code.
  - Area of active research for ~20 years.
  - Potentially much better coverage than dynamic analysis - examines all possible runs.
  - But typically also more false warnings.

• CHORD: static race and deadlock prediction for Java.
Outline

• Challenges for parallel testing.
• Random testing of parallel programs.
• Detecting and predicting parallel bugs.
• Active Random Testing of parallel programs.
• Conclusions.
Active Random Testing Overview

• **Problem**: Random testing can be very effective for parallel programs, but can miss many potential bugs.

• **Problem**: Predictive analyses find many bugs, but can have false warnings.
  - Time consuming and difficult to examine reported bugs and determine whether or not they are real.

• **Key Idea**: Combine them – use predictive analysis to find potential bugs, then **biased** random testing to actually create each bug.
**Active Random Testing Overview**

- **Key Idea**: Use predictive analysis to find potential bugs, then *biased* random testing to try to actually create each bug.

**Potential Data Races:**

- Race 1: (14, 19)
- Race N: (87, 92)

**Can lines 14 and 19 really race?**
• **Key Idea:** Use predictive analysis to find potential bugs, then biased random testing to try to actually create each bug.

**Potential Data Races:**

- Race 1: (14, 19)
- ...  
- Race N: (87, 92)

Can lines 14 and 19 really race?

100 random schedules

**Biased** to make it likely for lines 14 and 19 to race.

Only report data race to user if we see it in a real run.
Active Random Testing

• *CalFuzzer* is our extensible, open-source tool for active testing of Java programs.
  - For data races, atomicity bugs, and deadlocks.
  - *RaceFuzzer* is the active testing algorithm for data races – will show by example.

• *Thrille* for C + pthreads.
  - For data races.

• *And UPC-Thrille* for Unified Parallel C.
  - Part of the Berkeley UPC system by year’s end.
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

def sync foo(C x) {
    s1: g1();
s2: g2();
s3: g3();
s4: g4();
s5: x.f = 1;
}

def bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}

Run Predictive Analysis: Statement pair (s5,s6) are in race
RACEFUZZER using an example

Thread1
foo(o1);

t1: g1();
t2: g2();
t3: g3();
t4: g4();
t5: x.f = 1;
}

Thread2
bar(o1);

bar(C y) {
    s6: if (y.f==1)
        s7: ERROR;
    }

Thread3
foo(o2);

Run Predictive Analysis: Statement pair (s5,s6) are in race

8/17/10 Jacob Burnim Testing & Debugging: 36
(s5,s6) in race

Goal: Create a trace exhibiting the race
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Example Trace:
s1: g1();
s2: g2();
s3: g3();
s4: g4();
s5: o1.f = 1;
s6: if (o1.f==1)
s7: ERROR;
s4: g4();
s5: o2.f = 1;

Racing Statements Temporally Adjacent

(s5,s6) in race

Goal: Create a trace exhibiting the race
Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
(s5,s6) in race

sync foo(C x) {
    s1: g1();
    s2: g2();
    s3: g3();
    s4: g4();
    s5: x.f = 1;
}

bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}

Jacob Burnim

8/17/10
Thread 1
foo(o1);

sync foo(C x) {
    s1: g1();
    s2: g2();
    s3: g3();
    s4: g4();
    s5: x.f = 1;
}

Thread 2
bar(o1);

bar(C y) {
    s6: if (y.f == 1)
    s7: ERROR;
}

Thread 3
foo(o2);

Execution:
s1: g1();

(s5, s6) in race
Thread1
foo(o1);

sync foo(C x) {
    s1:  g1();
    s2:  g2();
    s3:  g3();
    s4:  g4();
    s5:  x.f = 1;
}

Thread2
bar(o1);

bar(C y) {
    s6:  if (y.f==1)
    s7:      ERROR;
}

Thread3
foo(o2);

Execution:
s1:  g1();
Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();

sync foo(C x) {
    s1: g1()
    s2: g2();
    s3: g3();
    s4: g4();
    s5: x.f = 1;
}

bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}

(s5,s6) in race
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1(); s1: g1();

(sync foo(C x) {
s1: g1()

s2: g2();

s3: g3();

s4: g4();

s5: x.f = 1;
})

bar(C y) {
s6: if (y.f==1)

s7: ERROR;
}

(s5,s6) in race
RACEFUZZER using an example

Thread 1

foo(o1);

sync foo(C x) {
  s1: g1()
  s2: g2()
  s3: g3()
  s4: g4()
  s5: x.f = 1;
}

Thread 2

bar(o1);

bar(C y) {
  s6: if (y.f==1)
  s7: ERROR;
}

Thread 3

foo(o2);

Execution:

s1: g1();

s1: g1();

s6: if (o1.f==1)

(s5,s6) in race
RACEFUZZER using an example

Thread 1
foo(o1);

Thread 2
bar(o1);

Thread 3
foo(o2);

Execution:
s1: g1();
s1: g1();
s6: if (o1.f==1)

(s5,s6) in race

bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s6: if (o1.f==1)

(s5,s6) in race

Postponed = { }

Thread1
sync foo(C x) {
s1: g1()
s2: g2();
s3: g3();
s4: g4();
s5: x.f = 1;
}

Thread2
bar(C y) {
s6: if (y.f==1)
s7: ERROR;
}

Thread3
foo(o2);

8/17/10 Jacob Burnim Testing & Debugging: 46
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s6: if (o1.f==1)
s6: if (o1.f==1)

(s5,s6) in race

Do not postpone if there is a deadlock

Postponed = { s6: if (o1.f==1) }
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1:  g1();
s1:  g1();

sync foo(C x) {
  s1:  g1()
  s2:  g2();
  s3:  g3();
  s4:  g4();
  s5:  x.f = 1;
}

bar(C y) {
  s6:  if (y.f==1)
  s7:   ERROR;
}

Postponed = {s6: if (o1.f==1) }
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s1: g1();
s2: g2();
s6: if (y.f==1)
s7: ERROR;
}

Postponed = {s6: if (o1.f==1) }
RACEFUZZER using an example

Thread1
foo(o1);

sync foo(C x) {
  s1: g1()
  s2: g2();
  s3: g3();
  s4: g4();
  s5: x.f = 1;
}

Thread2
bar(o1);

bar(C y) {
  s6: if (y.f==1)
  s7: ERROR;
}

Thread3
foo(o2);

Execution:
s1: g1();
s2: g2();
s3: g3();
s4: g4();
s5: x.f = 1;

Postponed = {s6: if (o1.f==1)}
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s2: g2();
s2: g2();
s3: g3();
s3: g3();
s4: g4();
s5: o2.f = 1;
s5: o2.f = 1;
s6: if (o1.f==1)

Postponed = {s6: if (o1.f==1)}
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s2: g2();
s2: g2();
s3: g3();
s3: g3();
s4: g4();
s5: o2.f = 1;
s6: if (y.f==1)
s7: ERROR;

Postponed = {s6: if (o1.f==1) }
(s5, s6) in race

Postponed = { s6: if (o1.f==1) }
RACEFUZZER using an example

Thread1
foo(o1);

sync foo(C x) {
    s1: g1();
    s2: g2();
    s3: g3();
    s4: g4();
    s5: x.f = 1;
}

Thread2
bar(o1);

bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();

Race?
NO

Postponed = {s6: if (o1.f==1)}

o1.f ≠ o2.f
RACEFUZZER using an example

Thread 1
foo(o1);

Thread 2
bar(o1);

Thread 3
foo(o2);

Execution:
s1: g1();
s1: g1();
s2: g2();
s2: g2();
s3: g3();
s3: g3();
s4: g4();
s4: g4();
s5: x.f = 1;
{s5, s6} in race

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2()
  s3:  g3()
  s4:  g4()
  s5:  x.f = 1;
}

Thread3
foo(o2);

Execution:
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }
**RACEFUZZER using an example**

Thread 1

```
foo(o1);
```

Thread 2

```
bar(o1);
```

Thread 3

```
foo(o2);
```

Execution:

```
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
```

Postponed = \{ **s6: if (o1.f==1), s5: o2.f = 1;** \}
RACEFUZZER using an example

Thread1
foo(o1);

Thread2
bar(o1);

Thread3
foo(o2);

Execution:
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
s5:  o1.f = 1;
s5:  o2.f = 1;

sync foo(C x) {
    s1:  g1()
    s2:  g2()
    s3:  g3()
    s4:  g4()
    s5:  x.f = 1;
}

bar(C y) {
    s6:  if (y.f==1)
    s7:  ERROR;
}

Postponed = {s6:  if (o1.f==1), s5:  o2.f = 1; }
Thread1                  Thread2                  Thread3                  Execution:
foo(o1);                bar(o1);                  foo(o2);                  s1:  g1();
s1:  g1();              s1:  g1();                s1:  g1();                s1:  g1();
s2:  g2();              s6:  if (y.f==1)              s2:  g2();                s2:  g2();
s2:  g2();              s7:  ERROR;                  s3:  g3();                s3:  g3();
s3:  g3();              }                                  s3:  g3();
s4:  g4();              s4:  g4();                  s4:  g4();                  s4:  g4();
s5:  x.f = 1;           s5:  o1.f = 1;               s5:  o1.f = 1;
}

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }
RACEFUZZER using an example

Thread1
foo(o1);

sync foo(C x) {
  s1:  g1()
  s2:  g2()
  s3:  g3()
  s4:  g4()
  s5:  x.f = 1;
}

Thread2
bar(o1);

bar(C y) {
  s6:  if (y.f==1)
  s7:     ERROR;
}

Thread3
foo(o2);

Execution:
s1:  g1();
s1:  g1();
s2:  g2();
s2:  g2();
s3:  g3();
s3:  g3();
s4:  g4();
s4:  g4();
s5:  o1.f = 1;

Postponed = {s6: if (o1.f==1), s5: o2.f = 1; }

(s5,s6) in race

Race?
YES
o1.f = o1.f

8/17/10 Jacob Burnim
Testing & Debugging: 60
RACEFUZZER using an example

(s5,s6) in race

Thread1
foo(o1);
sync foo(C x) {
    s1: g1();
s2: g2();
s3: g3();
s4: g4();
s5: x.f = 1;
}

Thread2
bar(o1);
bar(C y) {
    s6: if (y.f==1)
    s7: ERROR;
}

Thread3
foo(o2);

Execution:
s1: g1();
s1: g1();
s2: g2();
s2: g2();
s3: g3();
s3: g3();
s4: g4();
s4: g4();

Postponed = {s5: o2.f = 1; }
**RACEFUZZER using an example**

Thread1

foo(o1);

sync foo(C x) {
    s1:  g1()
    s2:  g2()
    s3:  g3()
    s4:  g4()
    s5:  x.f = 1;
}

Thread2

bar(o1);

bar(C y) {
    s6:  if (y.f==1)
        s7:     ERROR;
}

Thread3

foo(o2);

Execution:

s1:  g1();

s1:  g1();

s2:  g2();

s2:  g2();

s3:  g3();

s3:  g3();

s4:  g4();

s4:  g4();

s5:  o1.f = 1;

(s5,s6) in race

Postponed = {s5: o2.f = 1; }
RACEFUZZER using an example

![Diagram of multi-threaded execution with racing statements and postponed actions.](Image)

Thread 1
- `foo(o1);`
- `sync foo(C x) {` (s1: `g1();`
  - `s2: g2();`
  - `s3: g3();`
  - `s4: g4();`
  - `s5: x.f = 1;`)

Thread 2
- `bar(o1);`
- `bar(C y) {` (s6: `if (y.f==1)`
  - `s7: ERROR;`)

Thread 3
- `foo(o2);`

Execution:
- `s1: g1();`
- `s2: g2();`
- `s3: g3();`
- `s4: g4();`
- `s5: o1.f = 1;`
- `s6: if (o1.f==1)`

Postponed = {s5: `o2.f = 1;`}

Jacob Burnim

Testing & Debugging: 63

8/17/10
RACEFUZZER using an example

Thread 1
foo(o1);

Thread 2
bar(o1);

Thread 3
foo(o2);

Execution:
s1: g1();
s1: g1();
s2: g2();
s2: g2();
s3: g3();
s3: g3();
s4: g4();
s4: g4();
s5: o1.f = 1;
s5: o1.f = 1;
s6: if (o1.f==1)
s6: if (o1.f==1)
s7: ERROR;
s7: ERROR;

Postponed = {s5: o2.f = 1; }

Jacob Burnim
Testing & Debugging: 64
8/17/10
### RACEFUZZER using an example

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
<th>Thread3</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo(o1);</td>
<td>bar(o1);</td>
<td>foo(o2);</td>
<td></td>
</tr>
<tr>
<td>sync foo(C x) {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1: g1()</td>
<td></td>
<td>s1: g1()</td>
<td></td>
</tr>
<tr>
<td>s2: g2();</td>
<td></td>
<td>s2: g2();</td>
<td></td>
</tr>
<tr>
<td>s3: g3();</td>
<td></td>
<td>s3: g3();</td>
<td></td>
</tr>
<tr>
<td>s4: g4();</td>
<td></td>
<td>s4: g4();</td>
<td></td>
</tr>
<tr>
<td>s5: x.f = 1;</td>
<td></td>
<td>s5: o1.f = 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s6: if (o1.f==1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s7: ERROR;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Postponed = { }
Another RACEFUZZER Example

Thread1{
1: lock(L);
2: f1();
3: f2();
4: f3();
5: f4();
6: f5();
7: unlock(L);
8: if (x==0)
9: ERROR;
}

Thread2{
10: x = 1;
11: lock(L);
12: f6();
13: unlock(L);
}
Another RACEFUZZER Example

Thread1{
1: lock(L);
2: f1();
3: f2();
4: f3();
5: f4();
6: f5();
7: unlock(L);
8: if (x==0)
9:     ERROR;
}

Thread2{
10: x = 1;
11: lock(L);
12: f6();
13: unlock(L);
}

Racing Pair: (8,10)

This race would occur rarely under a normal or naively-random execution.
RaceFuzzer creates the race with high probability.
Another RACEFUZZER Example

Thread1{
1: lock(L);
2: f1();
3: f2();
4: f3();
5: f4();
6: f5();
7: unlock(L);
8: if (x==0)
9:    ERROR;
}

Thread2{
10:    x = 1;
11:    lock(L);
12:    f6();
13:    unlock(L);
}

Racing Pair: (8,10)  Postponed Set = {Thread2}

8/17/10 Jacob Burnim Testing & Debugging: 68
Another RACEFUZZER Example

Thread1{
1: lock(L);
2: f1();
3: f2();
4: f3();
5: f4();
6: f5();
7: unlock(L);
8: if (x==0)
9: ERROR;
}

Thread2{
10: x = 1;
11: lock(L);
12: f6();
13: unlock(L);
}
Another RACEFUZZER Example

Thread1{
  1: lock(L);
  2: f1();
  3: f2();
  4: f3();
  5: f4();
  6: f5();
  7: unlock(L);
  8: if (x==0)
     9:     ERROR;
}

Thread2{
  10: x = 1;
  11: lock(L);
  12: f6();
  13: unlock(L);
}

Jacob Burnim
Testing & Debugging: 70

8/17/10
Another RACEFUZZER Example

Thread1{
1: lock(L);
2: f1();
3: f2();
4: f3();
5: f4();
6: f5();
7: unlock(L);
8: if (x==0)
9:     ERROR;
}

Thread2{
10: x = 1;
11: lock(L);
12: f6();
13: unlock(L);
}

Hit error with 0.5 probability
Implementation

- RaceFuzzer: Part of CalFuzzer tool suite
- Instrument source using SOOT compiler framework
- Instrumentations are used to “hijack” the scheduler
  - Implement a custom scheduler
  - Run one thread at a time
  - Use semaphores to control threads
- Deadlock detector
  - Because we cannot instrument native method calls

```c
ins_lock(L1);
lock(L1);
ins_write(&X);
X=1;
unlock(L1);
ins_unlock(L1);
ins_lock(L1);
lock(L2);
Y=2;
unlock(L2);
ins_unlock(L1);
```
### Experimental Results

<table>
<thead>
<tr>
<th>Program Name</th>
<th>SLOC</th>
<th>Average Runtime in sec.</th>
<th># of Races</th>
<th># of Exceptions</th>
<th>Probability of hitting a race</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal</td>
<td>Hybrid</td>
<td>RF</td>
<td>Hybrid</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,352</td>
<td>2.07</td>
<td>&gt; 3600</td>
<td>42.37</td>
<td>59</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,924</td>
<td>3.25</td>
<td>&gt; 3600</td>
<td>3.81</td>
<td>2</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,619</td>
<td>3.48</td>
<td>&gt; 3600</td>
<td>6.44</td>
<td>5</td>
</tr>
<tr>
<td>cache4j</td>
<td>3,897</td>
<td>2.19</td>
<td>4.26</td>
<td>2.61</td>
<td>18</td>
</tr>
<tr>
<td>sor</td>
<td>17,689</td>
<td>0.16</td>
<td>0.35</td>
<td>0.23</td>
<td>8</td>
</tr>
<tr>
<td>hdec</td>
<td>29,948</td>
<td>1.10</td>
<td>1.35</td>
<td>1.11</td>
<td>9</td>
</tr>
<tr>
<td>weblech</td>
<td>35,175</td>
<td>0.91</td>
<td>1.92</td>
<td>1.36</td>
<td>27</td>
</tr>
<tr>
<td>jspider</td>
<td>64,933</td>
<td>4.79</td>
<td>4.88</td>
<td>4.81</td>
<td>29</td>
</tr>
<tr>
<td>jigsaw</td>
<td>381,348</td>
<td>-</td>
<td>-</td>
<td>0.81</td>
<td>547</td>
</tr>
<tr>
<td>vector 1.1</td>
<td>709</td>
<td>0.11</td>
<td>0.25</td>
<td>0.2</td>
<td>9</td>
</tr>
<tr>
<td>LinkedList</td>
<td>5979</td>
<td>0.16</td>
<td>0.26</td>
<td>0.22</td>
<td>12</td>
</tr>
<tr>
<td>ArrayList</td>
<td>5866</td>
<td>0.16</td>
<td>0.26</td>
<td>0.24</td>
<td>14</td>
</tr>
<tr>
<td>HashSet</td>
<td>7086</td>
<td>0.16</td>
<td>0.26</td>
<td>0.25</td>
<td>11</td>
</tr>
<tr>
<td>TreeSet</td>
<td>7532</td>
<td>0.17</td>
<td>0.26</td>
<td>0.24</td>
<td>13</td>
</tr>
</tbody>
</table>
Active Testing: Useful Features

- Classify real races from false alarms.
  - No false warnings.

- Inexpensive **replay** of a concurrent execution exhibiting a real race or other parallel bug

- Separate some harmful data races from benign races – i.e. whether or not the race leads to a crash or wrong output.

- Embarrassingly parallel.
  - Test different potential races / other bugs at the same time.
Active Testing: Limitations

• Not complete: can miss a real race.
  - Can only detect races that happen on the given test suite on some schedule.

• May not be able to separate all real races from false warnings.
  - Random scheduling may fail to create real race.

• May not be able to separate harmful races from benign races.
  - If error behavior not seen in random runs.

• Program is run sequentially during testing.
Active Testing Summary

• Combines benefits of random testing and predictive analysis.
  - Random testing amazingly effective in practice.
  - Even more so when biased with information about predicted bugs.
  - Can replay executions for debugging.

• Available now for Java (CalFuzzer) and Thrille (C + pthreads).
• UPC-Thrille for Unified Parallel C.
  - Part of the Berkeley UPC system by year’s end.
• Challenges for parallel testing.

• Random testing of parallel programs.

• Detecting and predicting parallel bugs.

• Active Random Testing of parallel programs.

• Conclusions.
Conclusions

• Many tools available right now to help find bugs in parallel software.
  - Data races, atomicity violations, deadlocks.

• But no silver bullet.
  - Have to carefully design how an application threads will coordinate and share/protect data.
  - Tools will help catch mistakes when the design is accidentally not followed.
  - Ad hoc parallelization likely to never be correct, even with these tools.
ANY QUESTIONS?
Pointers to Tools I


Pointers to Tools II


• CHORD (Static Race/Deadlock Detection for Java): http://code.google.com/p/jchord/
• CalFuzzer (Java):
  http://srl.cs.berkeley.edu/~ksen/calfuzzer/

• Thrille (C):
  http://github.com/nicholasjalbert/Thrille

• CHESS (C++/.NET Model Checking, Race Detection):

• Java Path Finder (Model Checking for Java):
  http://babelfish.arc.nasa.gov/trac/jpf