Program Testing and Analysis: Testing Concurrent Programs

Prof. Dr. Michael Pradel
Software Lab, TU Darmstadt
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

true  false  Something else
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

true  false  Something else
What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

Floating point numbers are represented with finite precision (not only in JavaScript)

true  false  Something else
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

0.30000000000000004
(due to rounding)

true false Something else
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
Why Bother with Concurrency?

- The free lunch provided by Moore’s law is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, multi-core processors became mainstream
  - Need concurrent programs to make full use of the hardware

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are "embarrassingly parallel"
Why Bother with Concurrency?

- The free lunch provided by Moore's law is over. CPU clock speeds stopped to increase around 2005.
- Instead, multi-core processors became mainstream.
- Need concurrent programs to make full use of the hardware.

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests.
  - Computations done on huge data often are "embarrassingly parallel."
Why Bother with Concurrency?

- The free lunch provided by Moore’s law is over
  - CPU clock speeds stopped to increase around 2005
  - Instead, multi-core processors became mainstream
  - Need concurrent programs to make full use of the hardware

- Many real-world problems are inherently concurrent, e.g.,
  - Servers must handle multiple concurrent requests
  - Computations done on huge data often are “embarrassingly parallel”
Concurrency Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in *actor concurrency model*, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via *locks* and *barriers*
Concurrency Styles

- **Message-passing**
  - Popular for large-scale scientific computing, e.g., MPI (message-passing interface)
  - Used in *actor concurrency model*, e.g., popular in Erlang and Scala
  - No shared memory (ideally), all communication via messages

- **Thread-based, shared memory**
  - Multiple concurrently executing threads
  - All threads access the same shared memory
  - Synchronize via *locks* and *barriers*

*Focus of this lecture*
Example

```java
int a = 0, b = 0;
boolean r = false, t = false;

a = 1;
r = true;
t = r;
b = a;
```

What does this program mean?

→ Behavior depends on thread interleaving
\[ a = b = 0; \quad r = t = \text{false} \]

\[ a = 1 \]
\[ r = \text{true} \]
\[ t = r \]
\[ b = a \]
\[ r = \text{true} \]
\[ t = f a l s e \]
\[ b = 1 \]
\[ t = \text{false} \]
\[ b = 1 \]

\[ t = \text{false} \]
\[ b = 1 \]

\[ t = \text{true} \] implies \[ b = 1 \]
Sequential Consistency

Assumption made here:
Programs execute under **sequential consistency**

- **Program order** is preserved: Each thread’s instructions execute in the specified order
- Shared memory behaves like a global array: **Reads and writes** are done immediately

- We assume sequential consistency for the rest of the lecture
- Many real-world platforms have more complex semantics ("memory models")
What Can Go Wrong?

Common source of errors: Data races

- Two accesses to the same shared memory location
- At least one access is a write
- Ordering of accesses is non-deterministic
Example

// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
Example

// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Thread 1

Thread 2

Read

Write

Shared memory location
Example

```java
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```

3 races
Example

```cpp
// bank account
int balance = 10;

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
```

Quiz: What values can `balance` have after executing this code?
Example

// bank account
int balance = 10;

Thread 1

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

Thread 2

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;

Possible outcomes:
balance may be 3, 8, and 15

But: Only 8 is correct
Avoiding Data Races

Use **locks** to ensure that **accesses** to shared memory do not interfere

```c
int balance = 10;
Thread 1
acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);
```

```c
Thread 2
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);
```
Avoiding Data Races

Use **locks** to ensure that **accesses to shared memory do not interfere**

```c
int balance = 10;
acquire(L);
int tmp1 = balance;
balance = tmp1 + 5;
release(L);
acquire(L);
int tmp2 = balance;
balance = tmp2 - 7;
release(L);
```

**Thread 1**

**Thread 2**

**Same lock ⇒ Mutually exclusive critical sections**
Avoiding Data Races

Use **locks** to ensure that accesses to shared memory **do not interfere**

```java
int balance = 10;

synchronized (L) {
    int tmp1 = balance;
    balance = tmp1 + 5;
}

synchronized (L) {
    int tmp2 = balance;
    balance = tmp2 - 7;
}
```

(Java syntax)
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
Eraser: Data Race Detection

- Basic idea: Look for "unprotected" accesses to shared memory

- Assumption: All accesses to a shared memory location $v$ should happen while holding the same lock $L$
  \[ \rightarrow \text{Consistent locking discipline} \]

- Dynamic analysis that monitors all lock acquisitions, lock releases, and accesses of shared memory locations
Lockset Algorithm (Simple Form)

- Let $\text{locksHeld}(t)$ be the set of locks held by thread $t$
- For each shared memory location $v$, initialize $C(v)$ to the set of all locks
- On each access to $v$ by thread $t$
  - Set $C(v) := C(v) \cap \text{locksHeld}(t)$
  - If $C(v) = \emptyset$, issue a warning
Lockset Algorithm (Simple Form)

- Let \( \text{locksHeld}(t) \) be the set of locks held by thread \( t \)

- For each shared memory location \( v \), initialize \( C(v) \) to the set of all locks

- On each access to \( v \) by thread \( t \)
  - Set \( C(v) \leftarrow C(v) \cap \text{locksHeld}(t) \)
  - If \( C(v) = \emptyset \), issue a warning
\[ balance = 10 \]

\begin{align*}
\text{acquire (L1)} & \quad \text{tmp1 = balance} \\
\text{balance = tmp1 + a} & \\
\text{release (L1)} & \\
\text{tmp2 = balance} & \\
\text{acquire (L2)} & \\
\text{balance = tmp2 - b} & \\
\text{release (L2)} & \\
\end{align*}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Locks Held</th>
<th>( C(balance) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = 10</td>
<td>{}\</td>
<td></td>
</tr>
<tr>
<td>acquire (L1)</td>
<td>{L1}</td>
<td>{L1, L2}</td>
</tr>
<tr>
<td>tmp1 = balance</td>
<td>{L1}</td>
<td>{}</td>
</tr>
<tr>
<td>balance = tmp1 + a</td>
<td>{}\</td>
<td></td>
</tr>
<tr>
<td>release (L1)</td>
<td>{}\</td>
<td></td>
</tr>
<tr>
<td>tmp2 = balance</td>
<td>{L2}</td>
<td>{}</td>
</tr>
<tr>
<td>acquire (L2)</td>
<td>{L2}</td>
<td>{}</td>
</tr>
<tr>
<td>balance = tmp2 - b</td>
<td>{}\</td>
<td>{} \rightarrow \text{warning}</td>
</tr>
<tr>
<td>release (L2)</td>
<td>{}\</td>
<td></td>
</tr>
</tbody>
</table>

\[ \]
Simple Lockset is Too Strict

Simple lockset algorithm produces **false positives** for

- variables initialized without locks held
- read-shared data read without locks held
- read-write locking mechanisms
  (producer-consumer style)
Refining the Lockset Algorithm

- Keep state of each shared memory location
- Issue warnings only in the Shared-modified state

Diagram:
- Virgin
- Exclusive
- Shared
- Shared-modified

Arrows indicate transitions:
- Read/write by 1st thread
- Write by 2nd thread
- Read by 2nd thread
- Write
Summary: Eraser

- **Dynamic analysis** to detect data races
- **Assumes** consistent locking discipline
- **Limitations**
  - May report false positives when locks are acquired inconsistently but correctly
  - May miss data races because it does not consider all possible interleavings
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
Thread Safety

- Popular way to encapsulate the challenges of concurrent programming: Thread-safe classes
- Class ensures correct synchronization
- Clients can use instances as if they were alone
- Rest of program can treat implementation of thread-safe class as a blackbox
Thread Safety (2)

“behaves correctly when accessed from multiple threads ... with no additional synchronization ... (in the) calling code”

“operations ... behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads”

StringBuffer API documentation, JDK 6
Example from JDK

```java
StringBuffer b = new StringBuffer()
    b.append("a")
    b.append("b")
    b.append("c")
    b.append("b")
```

Thread 1

Thread 2
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a")
b.append("b")
b.append("c")
```

Quiz: What can be the content of `b` if `StringBuffer` is thread-safe?
Example from JDK

```java
StringBuffer b = new StringBuffer()

Thread 1
b.append("a")

Thread 2
b.append("c")
b.append("b")

"abc" ✓
"cab" ✓
"acb" ✓
"ac" ✗
"bac" ✗
```
Example from JDK

```java
StringBuffer b = new StringBuffer()

b.append("a")
Thread 1
b.append("b")
Thread 2
b.append("c")

"abc" ✓ "cab" ✓ "acb" ✓ "ac" ✗ "bac" ✗
```
Example from JDK

```java
StringBuffer b = new StringBuffer()
    .append("a")
    .append("b")
    .append("c");
```

Thread 1: 
- `b.append("a")`
- `b.append("b")`
- `b.append("c")`

Thread 2: 
- `b.append("c")`

Results:
- "abc" ✓
- "cab" ✓
- "acb" ✓
- "ac" X
- "bac" X
Example from JDK

```java
StringBuffer b = new StringBuffer()
b.append("a")
b.append("b")
b.append("c")
```

![Diagram showing thread operations on StringBuffer](image)

- Thread 1:
  - `b.append("a")`
  - `b.append("b")`
- Thread 2:
  - `b.append("c")`

Possible states:
- "abc" ✓
- "cab" ✓
- "acb" ✓
- "ac" ✗
- "bac" ✗
Testing Thread-Safe Classes

- Correctness of program relies on thread safety of specific classes
- But: What if the class is actually not thread-safe?

- ConTeGe = Concurrent Test Generator
  - Creates multi-threaded unit tests
  - Detects thread safety violations by comparing concurrent behavior against linearizations
Example Bug from JDK

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Confirmed as bug: Issue #7100996
ConTeGe

Class under test (CUT) → Generate a concurrent test → Execute

Thread safety oracle → Bug
ConTeGe

Class under test (CUT) →

Generate a concurrent test

Execute

Thread safety oracle

Bug
ConTeGe

Class under test (CUT) → Generate a concurrent test

 Execute
 Thread safety oracle → Bug
Generating Concurrent Tests

Example:

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Thread 1  Thread 2
Generating Concurrent Tests

Example:

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
```

Sequential prefix:
Create and set up CUT instance

Thread 1

Thread 2

b.insert(1, b)    b.deleteCharAt(1)
Generating Concurrent Tests

Example:

StringBuffer b = new StringBuffer()
b.append("abc")

Thread 1
b.insert(1, b)

Thread 2
b.deleteCharAt(1)

Concurrent suffixes: Use shared CUT instance
Test Generation Algorithm

1. Create prefix
   - Instantiate CUT
   - Call methods

2. Create suffixes for prefix
   - Call methods on shared CUT instance

3. Prefix + two suffixes = test

Selection of methods similar to feedback-directed test generation
Creating a Prefix

1. Create prefix

- Instantiate CUT
- Call methods
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

Randomly select a constructor

```java
StringBuffer b = new StringBuffer();
```
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer();
```

After adding a call: Execute
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer();
b.append(/* String */);
```

Randomly select a method
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating a Prefix

1. Create prefix
   - Instantiate CUT
   - Call methods

```java
StringBuffer b = new StringBuffer();
b.append("abc")
```

After adding a call:
Execute
Creating a Prefix

1. Create prefix
   ■ Instantiate CUT
   ■ Call methods

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```
Creating Suffixes

2. Create suffixes for prefix
   - Call methods on shared CUT instance
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

```java
b.insert(/* int */ , /* CharSequence */)
```
Creating Suffixes

2. Create suffixes for prefix
- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value

```java
b.insert(/* int */ , /* CharSequence */)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(-5, b)
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(-5, b)
```

After adding a call:
Execute
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(/* int */ , /* CharSequence */)
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
```

Arguments:
- a) Take available object
- b) Call method returning required type
- c) Random value
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
```

After adding a call:
Execute
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")

b.insert(1, b)  // b.deleteCharAt(1)
```
Creating Suffixes

2. Create suffixes for prefix

- Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer();
b.append("abc")

b.insert(1, b) // b.deleteCharAt(1)
```

After adding a call: Execute
Creating Suffixes

2. Create suffixes for prefix
   - Call methods on shared CUT instance

```java
StringBuffer b = new StringBuffer()
b.append("abc")
```

```java
b.insert(1, b)   b.deleteCharAt(1)
```
Creating a Test

3. Prefix + two suffixes = test

```java
StringBuffer b = new StringBuffer()
b.append("abc")
b.insert(1, b)
b.deleteCharAt(1)
```

Spawn new thread for each suffix

Thread 1

Thread 2

b.insert(1, b)  b.deleteCharAt(1)
Approach

Class under test (CUT) →

Generate a concurrent test

Execute

Thread safety oracle

→ Bug
Approach

Class under test (CUT) → Generate a concurrent test → Execute → Thread safety oracle → Bug
Thread Safety Oracle

Does the test execution expose a thread safety violation?

- Focus on exceptions and deadlocks
- Compare concurrent execution to linearizations
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
Linearizations

- Put all calls into one thread
- Preserve order of calls within a thread
The Oracle

Execute concurrently

Exception or deadlock?

Yes

Execute linearization

Same failure?

Yes

No

No

All linearizations checked

Thread safety violation
Example

```java
StringBuffer b = new StringBuffer()
b.append("abc")

Thread 1
b.insert(1, b)   b.deleteCharAt(1)

Thread 2

StringBuffer b = ..
b.append("abc")
b.insert(1, b)   b.deleteCharAt(1)
```

```java
StringBuffer b = ..
b.append("abc")
b.deleteCharAt(1)
```

```java
StringBuffer b = ..
b.insert(1, b)
```
Example

StringBuffer b = new StringBuffer()
"abc"
b.insert(1, b)    b.deleteCharAt(1)

Thread 1    Thread 2

Thread safety violation

StringBuffer b = ..
"abc"
b.insert(1, b)    b.deleteCharAt(1)

StringBuffer b = ..
"abc"
b.insert(1, b)    b.deleteCharAt(1)
Properties of the Oracle

Sound but incomplete *

- All reported violations are real
- Cannot guarantee thread safety

Independent of bug type

- Data races
- Atomicity violations
- Deadlocks

* with respect to incorrectness
Implementation & Results

- Implemented for Java classes
- Applied to popular thread-safe classes from JDK, Apache libraries, etc.
- Found 15 concurrency bugs, including previously unknown problems in JDK
- Takes between several seconds and several hours (worst-case: 19 hours)
  - Recent master thesis, published at ICSE’17: Worst-case time reduced to several minutes
Outline

1. Introduction
2. Dynamic Data Race Detection
3. Testing Thread-Safe Classes
4. Exploring Interleavings

Mostly based on these papers:

- *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, Savage et al., ACM TOCS, 1997
- *Finding and Reproducing Heisenbugs in Concurrent Programs*, Musuvathi et al., USENIX 2008
Scheduling Non-Determinism

- A single program executed with a single input may have many different interleavings.
- Scheduler decides interleavings non-deterministically.
- Some interleavings may expose bugs, others execute correctly ("Heisenbugs").
- Challenge: How to explore different interleavings? How to detect buggy interleavings?
CHESS in a Nutshell

■ A user mode scheduler that controls all scheduling non-determinism

■ Guarantees:
  □ Every program run takes a new thread interleaving
  □ Can reproduce the interleaving for every run

■ Systematic but non-exhaustive exploration of the set of possible interleavings
Tree of Interleavings

- **Search space** of possible interleavings: Represent as a **tree**
- Node = points of **scheduling decision**
- Edge = decisions taken
- Each **path** = one **possible schedule**
Example

// bank account
int balance = 10;

Thread 1

// deposit money
int tmp1 = balance;
balance = tmp1 + 5;

Thread 2

// withdraw money
int tmp2 = balance;
balance = tmp2 - 7;
Scheduling tree

T1
1/2

T2
1/2

(0, 0)

(1, 0)

(2, 0)

(2, 1)

(2, 2)

(1, 1)

(2, 1)

(2, 2)

(1, 1)

(2, 1)

(2, 2)

(0, 1)

(2, 1)

(2, 2)

(0, 2)

(0, 2)

(2, 2)

(2, 2)

(2, 2)

(2, 2)

(2, 2)
State Space Explosion

- Number of interleavings: $O(n^n \cdot k)$
- Exponential in both $n$ and $k$
  - Typically: $n < 10$, $k > 100$
- Exploring all interleavings does not scale to large programs (i.e., large $k$)
Preemption Bounding

- Limit exploration to schedules with a small number \( c \) of preemptions
  - Preemption = Context switches forced by the scheduler
- Number of schedules: \( \mathcal{O}((n^2 \cdot k)^c \cdot n!) \)
  - Exponential in \( c \) and \( n \), but not in \( k \)
- Based on empirical observation: Most concurrency bugs can be triggered with few (< 2) interleavings
Implementation and Results

- Implemented via **binary instrumentation**
- Applied to eight mid-size and large systems (up to 175K lines of code),
- Found a total of **27 bugs**
- Major benefit over stress testing: Once a failure is detected, can **easily reproduce and debug it**
Summary

- Concurrent programming is inevitable
- Writing correct concurrent programs is hard
- Techniques to detect concurrency bugs
  - Dynamic data race detection
  - Test generation and thread safety checking
  - Systematic exploration of interleavings