Analysis Techniques to Detect Concurrency Errors

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College
Lecture Goals

• Enforcing concurrency properties
  - facilitates reasoning about correctness
  - race freedom, atomicity, determinism, cooperability

• Static and dynamic analyses
  - design space
  - implementation techniques
  - limitations

• Open research questions
Concurrent Programming Models

- **Shared memory** and explicit threads / sync

**Others**
- message passing, transactions, ...

Unshared: locals and control flow

Shared: objects and static fields
Deterministic Parallelism
Non-Deterministic Concurrency

Bank Server
Open Research Problems

• Making concurrency/parallelism readily accessible to all programmers

• Developing programming models beyond shared memory

• How to write efficient multithreaded code

• How to write correct multithreaded code
Thread Interference: Data Races

- Concurrent conflicting accesses
  - Two threads read/write, write/read, or write/write the same location without intervening synchronization

Thread A

...  
t1 = bal;
bal = t1 + 10;
...

Thread B

...  
t2 = bal;
bal = t2 - 10;
...
Thread Interference: Data Races

- Concurrent conflicting accesses
  - Two threads read/write, write/read, or write/write the same location without intervening synchronization

Thread A

...  
\[ t1 = \text{bal}; \]
\[ \text{bal} = t1 + 10; \]
...

Thread B

...  
\[ t2 = \text{bal}; \]
\[ \text{bal} = t2 - 10; \]
...

Thread A

\[ t1 = \text{bal} \]
\[ \text{bal} = t1 + 10 \]

Thread B

\[ t2 = \text{bal} \]
\[ \text{bal} = t2 - 10 \]
Thread Interference: Atomicity Violations

Thread A
...
acq(m);
t1 = bal;
rel(m);

acq(m);
bal = t1 + 10;
rel(m);

Thread B
...
acq(m);
bal = 0
rel(m);

Thread A

acq(m)
t1 = bal
rel(m)

Thread B

acq(m)
bal = t1 + 10
rel(m)
Thread Interference: Ordering Violations

Thread A
...
t = null;
fork(Thread B)
t = new Task()

Thread B
... 
t.perform();
...

Thread A
...
t = null
fork(Thread B)
t = new Task()

Thread B
...
t.perform()
Thread Interference: Unintended Sharing

```c
void work() {
    static int local = 0;
    local++;
    ...
}
```

Thread A
work();

Thread B
work();

```
Thread A
  tl = local
  local = tl+1

Thread B
  t2 = local
  local = t2+1
```
Thread Interference: Deadlock

class Account {
    int bal;
    synchronized void deposit(int n) {
        bal = bal + n;
    }

    synchronized void transfer(Account other, int n) {
        other.deposit(n);
        this.deposit(-n);
    }
}

Thread A
a.transfer(b,10);

Thread B
b.transfer(a,10);
Data Race Detection

- Atomicity violations
- Ordering violations
- Unintended sharing
- Deadlocks and livelocks

races are often a symptom of these errors
Thread Interference: Atomicity Violation

Thread A

...  
t1 = bal;
bal = t1 + 10;
...

Thread B

...  
t2 = bal;
bal = t2 - 10;
...

Thread A

Thread B

\[
\begin{align*}
t1 &= bal \\
bal &= t1 + 10 \\
t2 &= bal \\
bal &= t2 - 10
\end{align*}
\]
Thread Interference: Ordering Violations

Thread A

... 
t = null;
fork(Thread B)
t = new Task()

Thread B

t.perform();
...

Thread A

t = null
fork(Thread B)
t = new Task()

Thread B

t.perform()
Thread Interference: Unintended Sharing

```c
void work() {
    static int local = 0;
    local++;
    ...
}
```

Thread A
```
work();
```

Thread B
```
work();
```

- Thread A:
  - `t1 = local`
  - `local = t1+1`
- Thread B:
  - `t2 = local`
  - `local = t2+1`
Are All Race Conditions Errors?

• Implementing flag synchronization

```java
boolean done = false;

Thread A
x = 1;
done = true;

Thread B
if (done) t = x;
```

• Implementing fast reads

```java
int bal = 0;

Thread A
synchronized (m) {
    bal = bal + n;
}

Thread B
t = bal;
```
Are All Race Conditions Errors?

- Implementing flag synchronization
  - Treated as “synchronization”
  - Documents potential sharing
  - Improves program semantics
  - In C++: std::atomic<> types

```c++
volatile boolean done = false;

Thread A
x = 1;
done = true;

Thread B
if (done) t = x;
```

- Implementing fast reads

```c++
volatile int bal = 0;

Thread A
synchronized (m) {
  bal = bal + n;
}
```

Thread B
```c++
  t = bal;
```
Data Races and Memory Models

- Each processor/core has a cache
- When do writes to $x$ become visible to other processors (threads)?
Memory Models

- **Sequential Consistency**
  - Operations by threads are interleaved in some global sequential order.
  - A read yields the value most recently written to that location according to this order.
  - Simple, intuitive
Java Example

```java
int x;
int y;
Initially x == y == 0;
```

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 10;</td>
<td>r1 = y;</td>
</tr>
<tr>
<td>y = 20;</td>
<td>r2 = x;</td>
</tr>
<tr>
<td></td>
<td>print r1 + r2;</td>
</tr>
</tbody>
</table>

What’s Printed?  30?  20?  10?  0?
Memory Models

- **Sequential Consistency**
  - Operations by threads are interleaved in some global sequential order.
  - A read yields the value most recently written to that location according to this order.

- **Relaxed Models (JMM, x86-TSO, etc.)**
  - Writes may be buffered in caches
  - More than one value written to $x$ may be visible
  - Necessary for hardware performance
  - (also enables compiler optimizations)
Example

```java
int x = 0;
boolean done = false;

Thread A
x = 10;
done = true;

while (!done) {
    print x;
}

Thread B
while (!done) {
    print x;
}
```

```java
int x = 0;
volatile boolean done = false;

Thread A
x = 10;
done = true;

while (!done) {
    print x;
}

Thread B
while (!done) {
    print x;
}
```
Why Look For Races?

• Programmers make errors leading to data races:
  - Missing locking
  - Missing "volatile" annotations
  - ...

• Must know about races to reason about any more sophisticated concurrency property

• Memory Model Guarantee:
  - Data-Race Freedom $\rightarrow$ Seq. Consistent Behavior
Data Race Detection

- Automated Tools to Find Data Races
  - Active area of research for > 20 years
  - More than 100 academic papers on the subject

- Key dimensions of the design space are not unique to data-race detection
  - type-checking
  - array-bounds
  - pointer errors
  - etc.
Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants

- Example Tools:
  - RCC/Java type-based
  - CHESS state exploration
  - ESC/Java "functional verification" (theorem proving-based)
Static Data Race Detection

• Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants

• Disadvantages of static:
  - Undecidable...
  - Tools produce “false positives” or “false negatives”
  - May be slow, require programmer annotations
  - May be hard to interpret results
Dynamic Data Race Detection

• Advantages
  - Can avoid “false positives”
  - No need for language extensions or sophisticated static analysis

• Disadvantages
  - Run-time overhead (5-20x for best tools)
  - Memory overhead for analysis state
  - Reasons only about observed executions
    ▪ sensitive to test coverage
    ▪ (some generalization possible...)
Dynamic Analysis Design Space

- **Soundness**
  - every actual data race is reported
- **Completeness**
  - all reported warnings are actually races
- **Coverage**
  - generalize to additional traces?
- **Overhead**
  - run-time slowdown
  - memory footprint
- **Programmer overhead**
Overview of Analysis Techniques

- Lamport's Happens-Before Relation [Lamport 78]
  - enables precise definition of data race

- Four points in design space
  1. LockSet
  2. Vector Clocks
  3. Hybrid LockSet/VC
  4. FastTrack
**Happens-Before**

- Event Ordering:
  - program order
  - synchronization order
  - transitivity

- Types of Data Races:
  - Write-Write
  - Write-Read
    - (write then read)
  - Read-Write
    - (read then write)
Dynamic Data Race Detection

- Compute partial order of operations
- Ensure conflicting operations are not unordered
- Sound & Complete
- (No Trace Generalization)

Happens Before
[Lamport 78]

Eraser
[SBN+ 97]
Dynamic Data Race Detection

- Enforce consistent locking discipline (each variable is protected by a lock)
- Unsound & Incomplete
- Some trace generalization

Happens Before
[Lamport 78]

Eraser
[SBN+ 97]
Approximating Happens-Before

- Track lockset for each memory location
  - LockSet(x): set of locks held on all accesses to location x

- If \( m \in \text{LockSet}(x) \):

  \[
  \begin{align*}
  &x = 0 \\
  &\ldots \\
  &\text{rel}(m) \\
  &\text{acq}(m) \\
  &\ldots \\
  &t = x
  \end{align*}
  \]

- If LockSet(x) is empty:

  \[
  \begin{align*}
  &x = 0 \\
  &\ldots \\
  &\text{Data Race} \\
  &\ldots \\
  &t = x
  \end{align*}
  \]
**Lockset Example**

**Thread A**

```java
synchronized(x) {
  synchronized(y) {
    o.f = 2;
  }
  o.f = 11;
}
```

**Thread B**

```java
synchronized(y) {
  o.f = 2;
}
```

- **First access to o.f:**

  ```java
  LockSet(o.f) := Held(curThread)
  = \{ x, y \}
  ```
Lockset Example

Thread A

synchronized(x) {
    synchronized(y) {
        o.f = 2;
    }
    o.f = 11;
}

Thread B

synchronized(y) {
    o.f = 2;
}

• Subsequent access to o.f:

\[
\text{LockSet}(o.f) := \text{LockSet}(o.f) \cap \text{Held(curThread)}
\]

\[
= \{ x, y \} \cap \{ x \} = \{ x \} 
\]
Lockset Example

Thread A
synchronized(x) {
    synchronized(y) {
        o.f = 2;
    }
    o.f = 11;
}

Thread B
synchronized(y) {
    o.f = 2;
}

• Subsequent access to o.f:

  LockSet(o.f) := LockSet(o.f) ∩ Held(curThread)
  = \{ x \} ∩ \{ y \} = \{ \}

  DATA RACE!
Lockset Properties

- Relatively good performance (slowdowns $< \sim 15x$)
- Sound:
  - No warnings $\rightarrow$ data-race-free execution
- Incomplete:
  - Warning $\nRightarrow$ data race on execution
  - thread-local data, read-shared data, etc
Per-Variable State Machine

- First thread: r/w
- Second thread: read
- Any thread: r/w

Diagram:
- Thread Local
- Shared-read/write Track lockset
- Read Shared

Transitions:
- First thread: r/w
- Second thread: read
- Any thread: r/w
- Second thread write
- Any thread write
Lockset Properties

- Extensions help reduce false alarms but introduce (rare) unsoundnesses.
- And still not complete...

```java
boolean ready = false;
int data = 0;

// Thread A
data = 42;
sync(m) {
    ready = true;
}

// Thread B
sync(m) {
    tmp = ready;
}
if(tmp)
    print(data)
```
Dynamic Data-Race Detection

- Vector Clocks [M 88]
- Goldilocks [EQT 07]
- DJIT+ [ISZ 99,PS 03]
- TRaDe [CB 01]
- Happens Before [Lamport 78]

- Barriers [PS 03]
- Initialization [vPG 01]
- Eraser [SBN+ 97]

Precision vs Cost diagram
Precise Happens-Before
$\text{rel}(m) \quad \text{acq}(m) \quad \text{rel}(m) \quad \text{acq}(m) \quad \text{rel}(m) \quad \text{vol} = 1 \quad \text{acq}(m) \quad \text{rel}(m)$

[Diagram showing a sequence of operations starting with $\text{rel}(m)$, followed by $\text{acq}(m)$, then $\text{rel}(m)$, and finally $\text{acq}(m)$ again, with each operation affecting the values in a sequence of boxes.]

$\text{rel}(m)$

$\text{acq}(m)$

$\text{rel}(m)$

$\text{acq}(m)$

$\text{rel}(m)$

$\text{vol} = 1$

$\text{acq}(m)$

$\text{rel}(m)$

$\text{tmp} = \text{vol}$

$\text{acq}(m)$
rel(m) → rel(m) → rel(m) → tmp = vol → acq(m) → vol = 1 → acq(m) → rel(m)
rel(m) → acq(m) → rel(m) → vol = 1 → acq(m) → rel(m)

tmp = vol

acq(m) → rel(m)
B-steps with B-time $\leq 1$ happen before A's next step
Write-Write Check: $W_x \subseteq VC_A$?

- $3 \ 0 \ \subseteq \ 4 \ 1$? Yes

Read-Write Check: $R_x \subseteq VC_A$?

- $0 \ 1 \ \subseteq \ 4 \ 1$? Yes

$O(n)$ time
<table>
<thead>
<tr>
<th>$\mathbf{VC}_A$</th>
<th>$\mathbf{VC}_B$</th>
<th>$\mathbf{L}_m$</th>
<th>$\mathbf{W}_x$</th>
<th>$\mathbf{R}_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1</td>
<td>2 8</td>
<td>2 1</td>
<td>3 0</td>
<td>0 1</td>
</tr>
<tr>
<td>4 1</td>
<td>2 8</td>
<td>2 1</td>
<td>4 0</td>
<td>0 1</td>
</tr>
</tbody>
</table>
Write-Read Check: $W_x \subseteq VC_A$?

No

$O(n)$ time
VectorClocks for Data-Race Detection

- **Sound**
  - No warnings ➔ data-race-free execution

- **Complete**
  - Warning ➔ data-race exists

- **Slow performance**
  - (slowdowns > 50x)
Dynamic Data-Race Detection

- Eraser [SBN+ 97]
- Happens Before [Lamport 78]
- Barriers [PS 03]
- Initialization [vPG 01]
- RaceTrack [YRC 05]
- MultiRace [PS 03]
- Hybrid Detector [OC 03]
- Acculock [XXZ 13]
- IFRit [ELCGB 12]
- Vector Clocks [M 88]
- Goldilocks [EQT 07]
- DJIT+ [ISZ 99,PS 03]
- TRaDe [CB 01]
- RaceTrack [YRC 05]
- MultiRace [PS 03]
- Hybrid Detector [OC 03]
- Acculock [XXZ 13]
- IFRit [ELCGB 12]
**Combined Approaches**

- **MultiRace [PS 03,07]**
  - Use LockSet for $x$
  - Switch to VC if LockSet becomes empty
  - (adaptive granularity as well)

- **RaceTrack [YRC 05]**
  - Use Locket for $x$ with extensions to Eraser state machine.
  - Use VCs to reason about fork/join and wait/notify
Dynamic Data-Race Detection

FastTrack
[Flanagan-Freund 09]

Happens Before
[Lamport 78]

Barriers
[PS 03]
Initialization
[vPG 01]

Eraser
[SBN+ 97]

Vector Clocks
[M 88]
Goldilocks
[EQT 07]
Djit+
[ISZ 99, PS 03]
TRaDe
[CB 01]

RaceTrack
[YRC 05]
MultiRace
[PS 03]

Hybrid Detector
[OC 03]
Acculock
[XXZ 13]
IFRit
[ELCGB 12]

Vector Clocks
[M 88]
Goldilocks
[EQT 07]
Djit+
[ISZ 99, PS 03]
TRaDe
[CB 01]

RaceTrack
[YRC 05]
MultiRace
[PS 03]

Hybrid Detector
[OC 03]
Acculock
[XXZ 13]
IFRit
[ELCGB 12]
Dynamic Data-Race Detection

- FastTrack [Flanagan-Freund 09]
- Eraser [SBN+ 97]
- Barriers [PS 03]
- Initialization [vPG 01]
- ... 

Happens Before [Lamport 78]

- Vector Clocks [M 88]
- Goldilocks [EQT 07]
- ... 

- DJIT+ [ISZ 99, PS 03]
- TRaDe [CB 01]
- ... 

- RaceTrack [YRC 05]
- MultiRace [PS 03]
- Hybrid Detector [OC 03]
- Acculock [XXZ 13]
- IFRit [ELCGB 12]
- FastTrack [Flanagan-Freund 09]

• Design Criteria:
  - sound & complete
  (find at least 1st data race on each var)
  - efficient

• Insight:
  • HB relation is a partial order
  • But all accesses to a var are almost always totally ordered
<table>
<thead>
<tr>
<th>$VC_A$</th>
<th>$VC_B$</th>
<th>$L_m$</th>
<th>$W_x$</th>
<th>$R_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 1</td>
<td>2 8</td>
<td>2 1</td>
<td>3 0</td>
<td>0 1</td>
</tr>
</tbody>
</table>

Write-Write Check: $W_x \subseteq VC_A$?

3 0 $\subseteq$ 4 1 ? Yes

Read-Write Check: $R_x \subseteq VC_A$?

0 1 $\subseteq$ 4 1 ? Yes

$O(n)$ time
Write-Write and Write-Read Data Races

Thread A          Thread B          Thread C          Thread D

\( x = 0 \)        \( x = 1 \)        \( \text{read } x \)        \( x = 3 \)

\( O(n) \)
No Data Races Yet: Writes Totally Ordered

Thread A

x = 0

Thread B

x = 1

Thread C

read x

Thread D

x = 3

Complexity: $O(n)$
No Data Races Yet: Writes Totally Ordered

Thread A  Thread B  Thread C  Thread D

x = 0  x = 1  x = 3  

read x

\(O(1)\)
$VC_A$

4 1

$x = 0$

$VC_B$

2 8

$L_m$

2 1

$W_x$

1@B

Write-Write Check: $W_x \subseteq VC_A$ ?

1@B \leq 4 1 ? Yes

$(1 \leq 1?)$

O(1) time

Last Write "Epoch"
\[
\begin{array}{c|c}
\text{VC}_A & \text{VC}_B \\
\hline
4 & 1 \\
\hline
5 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{L}_m & \text{W}_x \\
\hline
2 & 1 \\
\hline
4 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{L}_m & \text{W}_x \\
\hline
2 & 1 \\
\hline
4 & 1 \\
\end{array}
\]

1. \(x = 0\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

2. \(\text{rel}(m)\)
   - \(\text{VC}_B\) moves to \(\text{VC}_A\)

3. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

4. \(x = 1\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

5. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

6. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

7. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

8. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

9. \(\text{acq}(m)\)
   - \(\text{VC}_A\) moves to \(\text{VC}_B\)

10. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

11. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

12. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

13. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

14. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

15. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

16. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

17. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

18. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

19. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

20. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

21. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

22. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

23. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

24. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

25. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

26. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

27. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

28. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

29. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

30. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

31. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

32. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

33. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

34. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

35. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

36. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

37. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

38. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

39. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)

40. \(\text{acq}(m)\)
    - \(\text{VC}_A\) moves to \(\text{VC}_B\)
Write-Read Check: \( W_x \subseteq VC_A \) ?

\[
\begin{array}{c}
8@B \\ \leq \\
5 \\
\end{array}
\]

No

\((8 \leq 1?)\)

\(O(1)\) time
Read-Write Data Races -- Ordered Reads

Most common case: thread-local, lock-protected, ...

Thread A

Thread B

Thread C

Thread D

x = 2
Read-Write Data Races -- Unordered Reads

Thread A  Thread B  Thread C

\[ x = 0 \]

fork

\[ \text{read } x \]

\[ x = 2 \]
Read-Write Check: $R_x \subseteq VC_A$?

8 1 $\subseteq$ 8 0? No
Thread A  

Thread B 

Thread C  

Thread D  

read y  

read y  

y = 10  

\( O(n) \)
Thread A  
read y  

Thread B  
read y  

Thread C  

y = 10  

Thread D
Thread A

read y

Thread B

read y

Thread C

y = 10

Thread D

y = 3

O(n)
Thread A

read y

Thread B

read y

Thread C

y = 10

Thread D

y = 3

\(O(1)\)

Forget VC for \(R_x\) and switch back to "last read epoch"
Memory Usage

- FastTrack allocated ~200x fewer VCs

<table>
<thead>
<tr>
<th>Checker</th>
<th>Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic VC, DJIT+</td>
<td>7.9x</td>
</tr>
<tr>
<td>FastTrack</td>
<td>2.8x</td>
</tr>
<tr>
<td>Empty</td>
<td>2.0x</td>
</tr>
</tbody>
</table>

(Note: VCs for dead objects are garbage collected)

- Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05]
Precise Data Race Classification for Other Checkers

and ~40% reduction in false alarms in Atomizer...
Eclipse 3.4

• Scale
  - > 6,000 classes
  - 24 threads
  - custom sync. idioms

• Precision (tested 5 common tasks)
  - Eraser: ~1000 warnings
  - FastTrack: ~30 warnings

• Performance on compute-bound tasks
  - > 2x speed of other precise checkers
  - same as Eraser
Verifying Race Freedom with Types

class Ref {
    int i;
    void add(Ref r) {
        i = i + r.i;
    }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    sync(x,y) { x.add(y); }
    sync(x,y) { x.add(y); }
}
assert x.i == 6;

Property: Each shared variable must be protected by a lock.
Verifying Race Freedom with Types

```java
class Ref {
    int i guarded_by this;
    void add(Ref r) requires this, r {
        i = i + r.i;
    }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    sync(x,y) { x.add(y); }
    sync(x) { x.add(y); }
}
assert x.i == 6;
```

Property: Each shared variable must be protected by a lock.

Error: lock y not held
Client-Side Locking

class Ref<ghost g> {
    int i guarded_by g;
    void add(Ref<g> r) requires g {
        i = i + r.i;
    }
}

Object m = new Object();
Ref<m> x = new Ref<m>(0);
Ref<m> y = new Ref<m>(3);
parallel {
    sync(m) { x.add(y); }
    sync(m) { x.add(y); }
}
assert x.i == 6;
Static Race Detection In Practice

- Rcc/Java [Flanagan-Freund 00-06]
- Other Systems
  - Ownership types [Boyapati et al 01]
  - RacerX [Engler-Ashcraft 02]
  - Chord [Naik et al 06]
  - Object Use Graphs [vonPraun-Gross 03]

- Limitations
  - scalability
  - unsound or incomplete
Static Analysis to Optimize Dynamic Checks

Options for Skip checks:
1. on race-free access.
2. that are redundant.

In source code:

\[
\begin{align*}
&\ldots\quad t = x.\texttt{Check} ; \\
&\ldots\quad u = y.\texttt{NoCheck} ; \\
&\ldots
\end{align*}
\]
Release-Free Spans

- Sequence of ops by one thread
- No outgoing edges
  - eg: no releases, forks, waits, ...
- If B races with C then A races with C
- Race check on B is redundant
RedCard: Redundant Check Elimination [ECOOP 2013]

- Find accesses always touching memory previously accessed within current release-free span
- Remove checks on those accesses

```
sync(m) {
    t = x.f;
    t = x.f;
}
t = x.f
```

- No change in precision
  - No missed races
  - No spurious warnings
Other Uses of Similar Notions

- **Interference-Free Regions** [Effinger-Dean et al 11, 12]
  - compiler optimizations, imprecise race detection
- **Similar optimizations for specific race detection algorithms**
  - Eraser-based [vonPraun-Gross 02, Choi et al 03]
  - X10 task parallelism [Raman et al 10]
- **RedCard**
  - works with any precise race detector
  - more sophisticated (but expensive) analysis
  - extensions for additional forms of redundancy
**Available Paths Analysis**

- For each program point, compute Context
  - Available Paths: expressions describing memory previously accessed in current span

\[
\begin{align*}
\{ \} & \quad t = x.f \text{Check} ; \\
\{ x.f \} & \quad t = x.f \text{NoCheck} ; \\
\{ x.f \} & \quad \text{rel}(m) ; \\
\{ \} & \quad t = x.f \text{Check} ; \\
\{ x.f \} & \quad \\
\end{align*}
\]

(for simplicity, assume no distinction between reads and writes)
Must Aliases

- Include must-alias constraints in C

\[
\begin{align*}
\{} & \quad \xrightarrow{\text{Check}} \quad x = z.g \\
\{z.g, \quad x = z.g\} & \quad \xrightarrow{\text{NoCheck}} \quad y = z.g \\
\{z.g, \quad x = z.g, \quad y = z.g\} & \quad \xrightarrow{\text{Check}} \quad t1 = x.h \\
\{z.g, \quad x.h, \quad x = z.g, \quad y = z.g\} & \quad \xrightarrow{\text{NoCheck}} \quad t2 = y.h
\end{align*}
\]

- Implement via any sound decision proc. (Z3)
- Similar to type state tracking [Fink et al 08]
Redundant Array Accesses

```java
for (int i = 0; i < a.length; i++) {
    a[i] \textcolor{red}{\textbf{Check}} = ...;
}
for (int i = 0; i < a.length; i++) {
    a[i] \textcolor{blue}{\textbf{NoCheck}} = ...;
}
```

- **Context extensions**
  - Paths for array accesses
    - single: \[ a[i] \]
    - range: \[ \forall (i \in 0 \text{ to } n). a[i] \]
  - Linear inequalities
\[ i = 0; \]
\[ \text{while} \ (i < a\text{.length}) \ { \}
\[ \quad a[i]^{\text{Check}} = 0; \]
\[ \quad i = i + 1; \]
\[ \}
\]
\[ a[k]^{\text{NoCheck}} = 1; \]

\[ \forall (j \in 0 \text{ to } a\text{.length}).a[j] \]
\begin{equation}
\forall (j \in 0 \text{ to } i). a[j]
\end{equation}

Inferred Via Cartesian Predicate Abstraction

\([BMMR\ 01,\ FQ\ 02]\)
\[
\begin{align*}
&\text{i} = 0; \\
&\text{while (i < a.length) \{ } \\
&\quad \text{a[i]}^{\text{Check}} = 0; \\
&\quad \text{i} = \text{i} + 1; \\
&\quad \text{a[i']}^{\text{NoCheck}} = 1;
\end{align*}
\]
RedCard Implementation for Java

- WALA framework for Java bytecode [IBM]
  - Dataflow analysis over SSA-based CFGs
  - Z3 [deMoura-Bjørner 08] to reason about Contexts

- Infers and outputs list of "NoCheck" accesses

- Two Modes
  - Intra-procedural
  - Inter-procedural (O-CFA, CHA)

- Analysis Time: ~18 sec per KLOC
% of Run-time Accesses Checked

<table>
<thead>
<tr>
<th></th>
<th>FastTrack</th>
<th>RedCard</th>
</tr>
</thead>
<tbody>
<tr>
<td>colt</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>crypt</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>lufact</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>moldyn</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>montecarlo</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mtrt</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>raja</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>raytracer</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>sparse</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>series</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>sor</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>tsp</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>elevator</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>philo</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>hdec</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>jbb</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
Proxy Fields

- Field y has proxy field x if all spans accessing p.y also access p.x

  If p.y has race then p.x has race

- Label p.y as "NoCheck"

- Still identify all traces with data races

```java
class Point {
    private int x, y;

    void move() {
        this.x\textsuperscript{Check} = ...;
        this.y\textsuperscript{NoCheck} = ...;
    }

    int dot(Point o) {
        return
        this.x\textsuperscript{Check} * o.x\textsuperscript{Check}
        + this.y\textsuperscript{NoCheck} * o.y\textsuperscript{NoCheck};
    }

    int getX() {
        return this.x\textsuperscript{Check};
    }
}
```
Array Proxies

- Array element can be proxy for other elements

<table>
<thead>
<tr>
<th>a[0] is proxy for a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[i div 4] is proxy for a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

- RedCard identifies common array proxy patterns
- b[j] is "NoCheck" if b[j] has proxy other than itself
  - may-alias info about b computed by separate analysis
% of Run-time Accesses Checked

RedCard  RedCard+Proxy

colt  crypt  lufact  moldyn  montecarlo  mtrt  raja  raytracer  sparse  series  sor  tsp  elevator  philo  hdec  jbb  Geo. Mean
Geo. Mean Slowdown (x Base Time)

- FastTrack: 7.5
- RedCard: 7.1
- RedCard + Proxy: 5.7
Where To Go From Here?

- Static Race Checking Analysis
- Performance (goal is always-on precise detection...)
  - HW support
  - static-dynamic hybrid analyses
  - sampling
- Coverage
  - symbolic model checking, specialized schedulers
- Classify malignant/benign data races
  - which data races are most critical?
- How to respond to data races? warn/fail-fast/recover?
- Reproducing traces exhibiting rare data races
  - record and replay
- Generalization: reason about traces beyond the observed trace
Key References

Key References

Key References

Jumble: Diagnosing Bad Races

- FastTrack finds real race conditions
  - races correlated with defects
  - cause unintuitive behavior, especially on relaxed memory models
  - but some are intentional/benign...

- Which race conditions are real bugs?
  - that cause erroneous behaviors (crashes, etc)
  - and are not “benign race conditions”
Controlling Scheduling Non-Determinism

Large Concurrent Application

Input

racy read

Thread A
p = new Pt();
...
p = null;
... if p != null
p.draw();

(eg: CalFuzzer)
Adversarial Memory [PLDI 2010]

Adversarial memory exploits memory nondeterminism.

Racy read sees old value likely to crash application.

Complements schedule-based approaches, quite effective.
Adversarial Memory [PLDI 2010]

Large Concurrent Application

Input

Thread A

```
p = new Pt();
...
p = null;
```

Thread B

```
... if p != null
   p.draw();
```
Sequentially Consistent Memory Model

- Intuitive memory model
- Each read sees most recent write
- (No memory caches)

```
it x = 10;
x = 0;
fork{ if (x != 0) x = 50/x; }
x = 42;
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```

```
x = 10
x = 0
fork
x = 42
```
Jumble

Record:
- write buffer for racy vars
- happens-before relation

At each read:
- determine visible writes
- return old writes to crash app with higher probability than typical memory impl.

```c
int x = 10;
x = 0;
fork{
    if (x != 0) x = 50/x;
}
x = 42;

r = x
r != 0?

r = x
r = 50/r
x = r
```

- not visible
- visible
- visible

heuristically pick 0

division by zero
## Jumble Precision: failures out of 100 runs

<table>
<thead>
<tr>
<th>Benchmark: racy field</th>
<th>No Jumble</th>
<th>SC</th>
<th>Oldest</th>
<th>Oldest but diff</th>
<th>Random</th>
<th>Random but diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>montecarlo: DEBUG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mtrt: threadCount</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>point: p</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>point: x</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>52</td>
<td>32</td>
<td>30</td>
</tr>
<tr>
<td>point: y</td>
<td>0</td>
<td>0</td>
<td>48</td>
<td>53</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>jbb: elapsed_time</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>jbb: mode</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>95</td>
<td>98</td>
</tr>
<tr>
<td>raytracer: checksum1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>sor: arrays</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>lufact: arrays</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>moldyn: arrays</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>tsp: MinTourLen</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

- 27 racy fields (found with FastTrack)
- ran Jumble manually once for each field
- found 4 destructive races
Student Contributors

• Jaeheon Yi, UC Santa Cruz (now at Google)
• Caitlin Sadowski, UC Santa Cruz (now at Google)
• Tom Austin, UC Santa Cruz (now at San Jose State)
• Tim Disney, UC Santa Cruz

• Ben Wood, Williams College (now at Wellesley College)
• Diogenes Nunez, Williams College (now at Tufts)
• Antal Spector-Zabusky, Williams College (now at UPenn)
• James Wilcox, Williams College (now at UW)
• Parker Finch, Williams College
• Emma Harrington, Williams College
Approximating Redundancy

- Record execution trace
- Annotate accesses in source based on dynamic occurrences in trace.

```plaintext
sync(m) {
    t = x.f\textcolor{red}{\textit{NonRedundant}};
    t = x.f\textcolor{red}{\textit{Redundant}};
    ...
    t = y.f\textcolor{red}{\textit{Redundant}};
}
```

Check on this line is necessary at least once.

Check on this line is always redundant.
Approximating Redundancy

• Record execution trace
• Annotate accesses in source based on dynamic occurrences in trace.

sync(m) {
    t = x.f
    t = x.f
    ... 
    t = y.f
}

\textcolor{red}{t = x.f} \textcolor{green}{NonRedundant};
\textcolor{red}{t = x.f} \textcolor{green}{Redundant};
\textcolor{red}{...}
\textcolor{red}{t = y.f} \textcolor{green}{Redundant};
\textcolor{red}{t = x.f} \textcolor{green}{NonRedundant};

\textcolor{red}{sync(m) \{}
    t = x.f
    t = x.f
    ... 
    t = y.f
\textcolor{red}{\}}
\textcolor{red}{t = x.f} \textcolor{green}{Check};
\textcolor{red}{t = x.f} \textcolor{green}{NoCheck};
\textcolor{red}{...}
\textcolor{red}{t = y.f} \textcolor{green}{Check};
\textcolor{red}{t = x.f} \textcolor{green}{Check};

• Compare to RedCard annotations
  \textcolor{green}{\text{NoCheck}} \text{ Accesses } \subseteq \text{ Redundant Accesses}
Where To Go From Here?

- **Static Race Checking Analysis**
- **Performance** (goal is always-on precise detection...)
  - HW support
  - static-dynamic hybrid analyses
  - sampling
- **Coverage**
  - symbolic model checking, specialized schedulers
- **Classify malignant/benign data races**
  - which data races are most critical?
- **How to respond to data races?** warn/fail-fast/recover?
- **Reproducing traces exhibiting rare data races**
  - record and replay
- **Generalization**
  - reason about traces beyond the observed trace
Increasing Redundancy

• Unroll first iteration of loops [Choi et al 03]

```java
for (i = 0; i < N; i++)
    p.fCheck.m();
if (i < N) {
    p.fCheck.m();
    for (i = 1; i < N; i++)
        p.fNoCheck.m();
}
```

• Other transformations:
  - method specialization
  - redundant synchronization elimination
  - ...

% of Run-time Accesses Checked

- RedCard
- RedCard+Proxy
- RedCard+Proxy w/ Unrolling
Geo. Mean Slowdown (x Base Time)

- FastTrack: 7.5
- RedCard: 7.1
- RedCard + Proxy: 5.7
- RedCard + Proxy w/ Unrolling: 6.4