FastTrack: Efficient and Precise Dynamic Race Detection (+ identifying destructive races)

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College
Multithreaded programming is notoriously difficult, in part due to schedule-dependent behavior:
- race conditions, deadlocks, atomicity violations, ...
- difficult to detect, reproduce, or eliminate
Race Conditions

- Two threads access a shared variable without synchronization, and at least one thread does a write
- Very common

2003 Blackout ($6 Billion)

Therac-25
Dynamic Race Detection

- Compute partial order of operations
- Ensure conflicting access are not concurrent
- Sound & Complete

Happens Before
[Lamport 78]

Eraser
[SBN+ 97]
Dynamic Race Detection

- Track locks held on all accesses to var.
  - empty lock set implies possible race
- Unsound & Incomplete

Happens Before
[Lamport 78]

Eraser
[SBN+ 97]
Dynamic 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]
Dynamic Race Detection

- Vector Clocks [M 88]
- Vector Clocks [EQT 07]
- Vector Clocks [PS 03]

- RaceTrack [YRC 05]
- MultiRace [PS 03]
- Hybrid Race Detector [OC 03]
-...

- Barriers
- Initialization [PS 03]
- ...

- Eraser [SBN+ 97]

Happens Before
[Lamport 78]

Precision

Cost
Dynamic Race Detection

FastTrack

- Design Criteria:
  - sound
    (find at least 1st race on each var)
  - complete (no false alarms)
  - efficient
- Insight: Accesses to a var are *almost always totally ordered* in the Happens-Before relation
Happens-Before

- **Event Ordering:**
  - program order
  - synchronization order

- **Types of Races:**
  - **Write-Write**
  - **Write-Read**
    - (write before read)
  - **Read-Write**
    - (read before write)
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
\[ \begin{array}{c|c}
\text{VC}_A & \text{VC}_B \\
\hline
4 & 1 \\
5 & 1 \\
\hline
\end{array} \quad \begin{array}{c|c}
L_m & W_x & R_x \\
\hline
2 & 1 & 3 & 0 & 0 & 1 \\
4 & 1 & 4 & 0 & 0 & 1 \\
4 & 1 & 4 & 0 & 0 & 1 \\
\end{array} \]
Write-Read Check: $W_x \subseteq VC_A$?

No

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

Thread A  Thread B  Thread C  Thread D

\[ x = 0 \]
\[ x = 1 \]
\[ \text{read } x \]
\[ x = 3 \]

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

Thread A  Thread B  Thread C  Thread D

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

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

Thread A  Thread B  Thread C  Thread D

x = 0  x = 1  x = 1  x = 3

read x

O(1)
Write-Write Check: $W_x \sqsubseteq V_{C_A}$?

$1@B \leq 41$ ? Yes

(1 $\leq$ 1?)

$O(1)$ time
Write-Read Check: $W_x \subseteq VC_A$ ?

$8@B \leq 51$ ? No

$(8 \leq 1?)$ O(1) time
Read-Write Races -- Ordered Reads

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

Thread A  Thread B  Thread C  Thread D

read x

read x

read x

read x

x = 2

?’
Read-Write Races -- Unordered Reads

Thread A    Thread B    Thread C

x = 0

fork

read x

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

\[
\begin{array}{c|c|c|c|c|c|c|c}
    & & & \ & \ & \ & \ \\
    & & & \ & \ & \ & \ \\
    & & & \ & \ & \ & \ \\
    & & & \ & \ & \ & \ \\
    8 & 1 & \subseteq & 8 & 0 & \ ? & \text{No}
\end{array}
\]
Thread A: read x
Thread B: read x

Thread C: x = 2

O(n)
Thread A

Thread B

Thread C

Thread D

read x

read x

x = 2
Thread A  Thread B  Thread C  Thread D

read x  read x

x = 2  x = 2

x = 3 \( O(n) \)
Thread A

Thread B

Thread C

Thread D

read x

read x

\( x = 2 \)

\( x = 3 \)

\( O(1) \)

Forget VC for \( R_x \) and switch back to "last read epoch"
RoadRunner Architecture

Standard JVM

Instrumented Bytecode

Event Stream
A: acq(m)
A: read(x)
B: write(y)
A: rel(m)

Back-End Checker

Error: race on x...
Validation

• Six race condition checkers
  - all use RoadRunner
  - share common components (eg, VectorClock)
  - profiled and optimized

• Further optimization opportunities
  - unsound extensions, dynamic escape analysis, static analysis, implement inside JVM, hardware support, ...

• 15 Benchmarks
  - 250 KLOC
  - locks, wait/notify, fork/join, barriers, ...
Warnings

22 false positives
3 false negatives

- Eraser [SBN+ 97] 27
- MultiRace [PS 03] 5
- GoldiLocks [EQT 07] 3
- Basic VC [M 88] 8
- DJIT+ [PS 03] 8
- FastTrack 8
Slowdown (x Base Time)

- Empty: 4.1
- Eraser: 8.6
- MultiRace: 21.7
- Goldilocks: 31.6
- Basic VC: 89.8
- DJIT+: 20.2
- FastTrack: 8.5
O(n) Vector Clock Operations

![Graph showing performance comparison of different benchmarks](image)
96.4% of all ops are Reads/Writes

R/W ops requiring O(n) time:

<table>
<thead>
<tr>
<th></th>
<th>Basic VC</th>
<th>DJIT+</th>
<th>FastTrack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator</td>
<td>100%</td>
<td>26.0%</td>
<td>&lt;0.1%</td>
</tr>
</tbody>
</table>

96.4% of all operations are reads/writes, and among these, 96.4% are executed in O(n) time under different execution environments.
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>
</tbody>
</table>

(Note: VCs for dead objects can be garbage collected)

• Improvements
  - accordion clocks [CB 01]
  - analysis granularity [PS 03, YRC 05] (see paper)
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
Beyond Detecting Race Conditions

- FastTrack finds real race conditions
  - races correlated with defects
  - cause unintuitive behavior on relaxed memory

- Which race conditions are real bugs?
  - that cause erroneous behaviors (crashes, etc)
  - and are not “benign race conditions”
class Point {
    double x, y;
    static Point p;

    Point() { x = 1.0; y = 1.0; }

    static Point get() {
        Point t = p;
        if (t != null) return t;
        synchronized (Point.class) {
            if (p == null) p = new Point();
            return p;
        }
    }

    static double slope() {
        return get().x / get().y;
    }

    public static void main(String[] args) {
        fork { System.out.println( slope() ); }
        fork { System.out.println( slope() ); }
    }
}
Thread 0

\( p = \text{null} \)
\( px = 0 \)
\( py = 0 \)

fork 1,2

Thread 1

read \( p \) // null
acquire
read \( p \) // null
\( p = \text{new Point} \)
\( px = 1 \)
\( py = 1 \)
release
read \( px \) // get 1
read \( py \) // get 1

read \( p \) // non-null
read \( px \) // ?

Thread 2
Thread 0

p = null
px = 0
py = 0
fork 1, 2

Thread 1

read p // null
acquire
read p // null
p = new Point
px = 1
py = 1
release
read px // get 1
read py // get 1

Thread 2

read p // null
read px // ?
read py // ?
Thread 0

Thread 1

Thread 2

Race: can return either write (mm non-determinism)

Typical JVM: mostly sequentially consistent

Adversarial memory
  - use heuristics to return older stale values
Adversarial Memory

- Record history of all writes (plus VCs) to racy variables
- At read
  - determine all visible writes legal under JMM
  - heuristically pick one likely to crash target program
- Six heuristics:
  - Sequentially consistent: return last write
  - Oldest: return “most stale” value
  - Oldest-but-different: never return same val twice
    - if (p != null) p.draw()
  - Random, Random-but-different
# Experimental Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Field</th>
<th>Erroneous Behavior Observation Rate (%)</th>
<th>Destructive Race?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>JUMBLE configurations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No Jumble</td>
<td>Sequentially Consistent</td>
</tr>
<tr>
<td>Figure 1</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Figure 2</td>
<td>p</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>p.x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>p.y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hedc</td>
<td>Task.thread</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MetaSearchResult.results</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MetaSearchResult.completed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MetaSearchResult.request</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Task.valid</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jbb</td>
<td>Company.elapsed.time</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Company.mode</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>Universal.UNIVERSAL_DEBUG</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mtrt</td>
<td>RayTracer.threadCount</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>JGFRayTracerBench.checksum1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>TspSolver.MinTourLen</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sor</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lufact</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>array index [0] and [1]</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>