Towards Efficient and Precise Concurrent Software Analysis

Cormac Flanagan
UC Santa Cruz

- Stephen Freund, Williams College
- 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 University)
- Tim Disney, UC Santa Cruz (now at Google)
- Dustin Rhodes (now at Google)
- 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
Multicore CPUs
Natural language

Programming language

Syntax
- ...

Semantics
- correctness
- modularity
- security
- testability
- ...

Multicore hardware
- threads
- shared memory
- preemptive scheduling
- relaxed memory models
Sequential Software: Deterministic

Large Sequential Application

Input
Multithreaded Software: Nondeterministic Preemptive Scheduling
Relaxed Memory Models

- Does each read see the “most recent” write?
  - Sequentially Consistent MM => Yes
  - Relaxed MM (JMM, x86-TSO, etc.) => No
Double NonDeterministic “Demons” of Multithreading

Preemptive scheduling

Relaxed memory model
Multiple Threads

\[
x++
\]

is a non-atomic read-modify-write

```
x = 0;
thread interference?
while (x < len) {
    thread interference?
    tmp = a[x];
    thread interference?
    b[x] = tmp;
    thread interference?
    x++;
    thread interference?
}
```

Single Thread

\[
x++
\]

```
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
```
Controlling Thread Interference
#1 Enforce Race Freedom
Controlling Thread Interference: #1 Enforce Race Freedom

- Race Condition
  two concurrent unsynchronized accesses, at least one write

Thread A
... t1 = bal; bal = t1 + 10; ...

Thread B
... t2 = bal; bal = t2 - 10; ...

Thread A
  t1 = bal
  bal = t1 + 10

Thread B
  t2 = bal
  bal = t2 - 10
Controlling Thread Interference: #1 Enforce Race Freedom

- Race Condition
  
  two concurrent unsynchronized accesses, at least one write

```
Thread A
...
t1 = bal;
bal = t1 + 10;
...

Thread B
...
t2 = bal;
bal = t2 - 10;
...
```
Controlling Thread Interference: #1 Enforce Race Freedom

- Many analyses to detect races
  - AAF'06, AS'04, AG'98, BR'01, DC'94, EA'03, G'03, NAW'06, VJL'07, PFH'06, PS'07, SBNSA'97, vPG'01, YRC'05, FF'09, CC'03, BCM'10

- Races are correlated to defects

- Theorem 1
  - Any race-free program behaves as if running on sequentially consistent memory model
Types For Race Freedom: java.util.Vector

class Vector {
    Object elementData[] guarded_by this;
    int elementCount guarded_by this;

    int lastIndexOf(Object o) {
        return lastIndexOf(o, elementCount - 1);
    }

    synchronized int lastIndexOf(Object o, int index) {
        ... RACE
        return lastIndexOf(o, elementCount - 1);
    }

    synchronized int lastIndexOf(Object o, int index) {
        ... IndexOutOfBoundsException
        ... RACE
    }

    ... [TOPLAS 2006]
Controlling Thread Interference
#2 Beyond Race Freedom
An Introduction to Programming with Threads

by Andrew D. Birrell

January 6, 1989

digital
Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301
Race Freedom is not Enough

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

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

Thread B
...
acq(m);
bal = bal - 10;
rel(m);

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

Thread B
acq(m)
bal = bal-10
rel(m)

acq(m)
bal = t1 + 10
rel(m)
Controlling Thread Interference: #2 Enforce Atomicity

Atomic method must behave as if it executed serially, without interleaved operations of other thread.

- sequential reasoning valid for atomic methods
- 90% of methods are atomic

```java
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```
Theory of Reduction [Lipton 76]

 Serializable blocks have the pattern: \( R^{*} [N] L^{*} \)
A Type System for Atomicity

• Theorem 2
  – Any well-typed program behaves *as if* each atomic method executes serially (without interleaved steps of other threads) [toplas’08]

• Many other analyses for atomicity
  • FFY’08, FF’04, FFLQ’08, WS’06, XBH’06, PLZ’09, RDFHLR’05, FM’08
A Type System for Atomicity

- Many analyses for atomicity
  - FFY’08, FF’04, FFLQ’08, WS’06, XBH’06, PLZ’09, RDFHLR’05, FM’08

- Including a type system for atomicity
  - TOPLAS’08

- Theorem 2
  - Any well-typed program behaves as if each atomic method executes serially, without interleaved steps of other threads
java.lang.StringBuffer

/**
   ... used by the compiler to implement the binary string concatenation operator ...

String buffers are safe for use by multiple threads. The methods are synchronized so that all the operations on any particular instance 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 involved.

*/

public atomic class StringBuffer { ... }
java.lang.StringBuffer is not Atomic

public atomic StringBuffer {
    private int count guarded_by this;
    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }

    public synchronized void append(StringBuffer sb){

        int len = sb.length();
        ...
        ...
        sb.getChars(...,len,...);
        ...
    }
}

• violates pattern \((R^*|N)L^*)\), append() is not atomic
Controlling Thread Interference
#3 Beyond Atomicity
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}

void busy_wait() {
    acq(m);
    thread interference?
    while (!test()) {
        thread interference?
        rel(m);
        thread interference?
        acq(m);
        thread interference?
        x++;
        thread interference?
    }
}

• ~90% of methods atomic
• Sequential reasoning

• ~10% of methods not atomic
• Pervasive interference
• Atomicity provides no help
• Local atomic blocks awkward
Controlling Thread Interference: #3 Explicit Yields

```c
atomic {
    ...
    ...
}
```

weird semantics

```c
atomic {
    ...
    ...
}
```

weird semantics

```c
atomic {
    ...
    ...
}
```

good semantics

```c
yield
```
Non-Preemptive Scheduling

• Context switches only at yields

• Clean semantics
  – Sequential reasoning valid by default ...
  – ... except where yields highlight thread interference

• Limitation: Uses only a single processor
Non-Preemptive Scheduler
- Sequential reasoning
- Except where yields indicate interference

Preemptive Scheduler
- Full performance
- No overhead

Flanagan ETAPS 2019
Non-Interference Design Space

<table>
<thead>
<tr>
<th>Policy Enforcement</th>
<th>Non-Interference Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>traditional sync + analysis</td>
<td>atomicity, serializability</td>
</tr>
<tr>
<td>new run-time systems</td>
<td>transactional memory</td>
</tr>
</tbody>
</table>

Transactional Memory, Larus & Rajwar, 2007
Automatic mutual exclusion, Isard & Birrell, HOTOS '07
**Multiple Threads**

\[
x++
\]

is a non-atomic read-modify-write

\[
x = 0;
while (x < len) {
    thread interference?
    tmp = a[x];
    thread interference?
    b[x] = tmp;
    thread interference?
    x++;
    thread interference?
}
\]

**Single Thread**

\[
x++
\]

\[
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
\]
Explicit Yields

\[
\begin{align*}
\text{x++ vs. } & \quad \text{yield; } \quad \text{vs. } \quad \text{x++} \\
\text{x = 0; } & \\
\text{while (x < len) { } } & \\
\quad & \quad \text{yield; } \\
\quad & \quad \text{tmp = a[x]; } \\
\quad & \quad \text{yield; } \\
\quad & \quad \text{b[x] = tmp; } \\
\quad & \quad \text{x++; } \\
\quad & \quad } \\
\end{align*}
\]

Flanagan ETAPS 2019
A Type System for Preemptive/non-preemptive equivalence

• Theorem 3
  – Any well-typed program behaves as if run on a non-preemptive scheduler (even when run on preemptive/multicore hardware)

• Other analyses
  • eg IB’07, YF’10, YSF’11, CCHRRRT’17
class StringBuffer {

    synchronized StringBuffer append(StringBuffer sb) {
        ...
        int len = sb.length();
        yield;
        ...
        // allocate space for len chars
        sb.getChars(0, len, value, index);
        return this;
    }

    ...
}

• Yields help programmers identify defects
  – difference is statistically significant
  – [Sadowski, Yi  PLATEAU 2010]
Review of Non-interference Specs

• Race freedom
  – code behaves as if on sequentially consistent memory model

• Atomicity
  – code behaves as if atomic methods executed serially (~90%)

• Yield-oriented programming
  – code behaves as if run on non-preemptive scheduler
  – sequential reasoning ok ...
  – ... except where yields indicate thread interference (1-10/KLOC)
  – http://users.soe.ucsc.edu/~cormac/coop.html
Analysis Tools for Non-Interference
Analysis Tools for Non-Interference

Static analysis
• observe syntax
• over-approximate behavior
• report all errors (theorems!)
• report (many?) false alarms

Dynamic analysis
• observe traces
• under-approximate behavior
• miss some errors
• can guarantee no false alarms
Precise Dynamic Race Detection

Overhead

DJIT+
[PS 07]
Dynamic Race Detection Overhead

class Point {
    int x, y;
    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }
    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i] = 0;
        }
    }
}

Object Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shadow Memory

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Races

- Happens-Before Relation [Lamport 78]

Thread A

```java
sync(lock) {
    b.f = 0;
}
```

Thread B

```java
sync(lock) {
    x = b.f;
}
```

Happens-Before Relation \cite{Lamport78}
Data Races

• Happens-Before Relation [Lamport 78]
• Data Race: unordered accesses

Thread A

```c
sync(lock) {
    b.f = 0;
}
```

Thread B

```c
sync(lock) {
    x = b.f;
}
```
Data Races

- Happens-Before Relation [Lamport 78]
- Data Race: unordered accesses

Thread A

\[
\text{sync(lock) \{ }
\]
\[
\text{b.f = 0; }
\]
\[
\text{\}
\]

Thread B

\[
\text{sync(lock) \{ }
\]
\[
\text{x = b.f; }
\]

Data Race

I won't distinguish reads vs. writes
Tracking the Happens-Before Relation

- Program Order
- Synchronization Order
Vector Clocks
[Mattern 88]
Vector Clock Checks

A

B

x=2

C

x=3

D

x=4

O(n)
Vector Clock Checks

A

x=1

B

x=2

C

x=3

D

x=4

O(1)!
Epoch Checks
class Point {
    int x,y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i]=0;
        }
    }
}
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i] = 0;
        }
    }
}
Precise Dynamic Race Detection

Overhead

DJIT+ [PS 07]  FastTrack [FF 09]
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i]=0;
        }
    }
}
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
        check(this.{x,y});
    }
}

static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
        check(a[i]); a[i]=0;
    }
    check(a[0..n-1]);
}
Precise Check Placement

• No Missed Races

```plaintext
sync(lock) {
    check(b.f)
    x = b.f;
    check(b.f)
}

    check(b.f)
    y = b.f;
    check(b.f)
```

• No False Alarms

```plaintext
sync(lock) {
    check(b.f)
    z = b.f;
    ...
}
```
Precise Check Placement

• No Missed Races
• Access must have a covering check between
  – previous release
  – next acquire

• No False Alarms

```plaintext
sync(lock) {
  x = b.f;
}

sync(lock) {
  y = b.f;
}

sync(lock) {
  z = b.f;
}
```

Flanagan ETAPS 2019
Precise Check Placement

- **No Missed Races**
  - Access must have a covering check between
    - previous release
    - next acquire

- **No False Alarms**
  - Check must have a legitimizing access between
    - previous acquire
    - next release
Precise Check Placement

- **No Missed Races**
  - Access must have a *covering check* between
    - previous release
    - next acquire

- **No False Alarms**
  - Check must have a *legitimizing access* between
    - previous acquire
    - next release
1. **Static BigFoot**
   - Fewer, bigger checks: check(a[0..n])
   - Intra-procedural dataflow analysis
   - WALA [IBM], Z3 [DB 08]

2. **Dynamic BigFoot**
   - Compress shadow state
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }
}

static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
        check(a[i]); a[i]=0;
    }
}
class Point {
    int x,y;
    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
        check(this.{x,y});
    }
}

static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
        check(a[i]); a[i]=0;
    }
    check(a[0..n-1]);
}
Static Object Shadow Compression

• Compress fields of class that always appear in check statements together

```java
... 
 sync(lock) { 
   pt.x = 1; 
   check(pt.x); 
   pt.y = 2; 
   check(pt.y); 
 } 
 ...
```
Static Object Shadow Compression

- Compress fields of class that always appear in check statements together

    ...  
    sync(lock) {
        pt.x = 1;
        pt.y = 2;
        check(pt.{x,y});
    }
    ...  
    check(b.{x,y});
    ...
    check(c.{x,y});
    ...

Flanagan ETAPS 2019
Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary
Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary

```java
sync(lock) {
    for(int i=0;i<n;i++) a[i]=0;
    check(a[0..n-1]);
}
```
Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary

```java
sync(lock) {
    for (int i = 0; i < n/2; i++) a[i] = 0;
    check(a[0..n/2-1]);
    for (int i = n/2; i < n; i++) a[i] = 0;
    check(a[n/2..n-1]);
}
```

- Buffer checks until release, as in ReoPlay [RB 99], DRD [D 14], ThreadSanitizer [SI 09]
Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary

```java
sync(lock) {
    for(int i=0; i<n/2; i++) a[i] = 0;
    check(a[0..n/2-1]);
}
```
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i] = 0;
        }
    }
}
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
        check(this.{x,y});
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i] = 0;
        }
        check(a[0..n-1]);
    }
}
class Point {
    int x, y;

    void move(int dx, int dy) {
        int tmp;
        check(this.x); tmp = this.x;
        check(this.x); this.x = tmp + dx;
        check(this.y); tmp = this.y;
        check(this.y); this.y = tmp + dy;
        check(this.{x,y});
    }

    static void clear(int[] a, int n) {
        for (int i = 0; i < n; i++) {
            check(a[i]); a[i]=0;
        }
        check(a[0..n-1]);
    }
}
BigFoot Eliminates Checks

Check Ratio $\frac{\text{# Checks}}{\text{# Accesses}}$

Lower is Better

- FastTrack: 1
- BigFoot: 0.43
Precise Dynamic Race Detection

Overhead

<table>
<thead>
<tr>
<th>System</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>DJIT+ [PS 07]</td>
<td>20.0x</td>
</tr>
<tr>
<td>FastTrack [FF 09]</td>
<td>6.0x</td>
</tr>
<tr>
<td>RedCard [FF 13]</td>
<td>5.0x</td>
</tr>
<tr>
<td>SlimState [WFFF 15]</td>
<td>5.0x</td>
</tr>
<tr>
<td>BigFoot [RFF 17]</td>
<td>0.0x</td>
</tr>
</tbody>
</table>
Summary

• Race freedom - as if sequentially consistent
• Atomic methods - as if executed serially
• Explicit yields - as if on non-preemptive scheduler

<table>
<thead>
<tr>
<th>Object Memory</th>
<th>DJIT+ 20x</th>
<th>FastTrack 7.3x</th>
<th>BigFoot 2.5x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>