Analysis Techniques to Detect Concurrency Errors (part 2)

Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College
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
Multithreading and Multicore CPUs

- Multithreaded programming is here. And difficult.
  - schedule-dependent behavior
  - race conditions, deadlocks, atomicity violations, ...
  - difficult to detect, reproduce, or eliminate
Static Techniques:
• Types for Race Detection
• Types for Atomicity

Dynamic Techniques:
• Data Races
• Atomicity / Serializability
• Cooperative Concurrency
RoadRunner Framework for Dynamic Concurrency Analyses [PASTE ‘10, github]

Tool API

abstract class Tool {
  void create(NewThreadEvent e)
  void acquire(AcquireEvent e)
  void release(ReleaseEvent e)
  void access(AccessEvent e)
  ...
}

class Copper extends Tool {
  - handlers for synchronization / access events
  - data to store about program state
}
Controlling Thread Interference

Static Techniques:
• Types for Race Detection
• Types for Atomicity

Dynamic Techniques:
• Data Races
• Atomicity / Serializability
• Cooperative Concurrency
Multiple Threads

```
x++
```

is a non-atomic read-modify-write

---

Single Thread

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

1. Inspect code
2. Identify where interference does not occur

```c
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
```
Controlling Thread Interference: #1 Manually w/ "Productivity Heuristic"

1. Assume no interference
2. Use sequential reasoning

- Works some of the time, but subtle bugs...
Controlling Thread Interference: #2 Enforce Race Freedom

- Race Conditions
  
  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 |
| t2 = bal |
| bal = t2 - 10 |
Controlling Thread Interference: #2 Enforce Race Freedom

- Race Conditions
  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: #2 Enforce Race Freedom

• Race Conditions
  two concurrent unsynchronized accesses, at least one write

• Races are correlated to defects
• Race-freedom ensures sequentially-consistent behavior
  – even on relaxed memory models

• But...
Controlling Thread Interference: #2 Enforce Race Freedom

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)
Controlling Thread Interference: #3 Enforce Atomicity

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

```c
void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```
Controlling Thread Interference: #3 Enforce Atomicity

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

```c
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```

• Can use sequential reasoning in atomic methods

• 90% of methods
Bohr: Static Analysis for Atomicity

- Extension of Java’s type system [TOPLAS’08]
- Input: Java code with
  - atomicity annotations
  - annotations describing protecting lock for fields
- Theorem: In any well-typed program, all paths through atomic methods are serializable
Theory of Reduction [Lipton 76]

Serializ able blocks have the pattern: \( R^* [N] L^* \)
Examples

```java
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        bal = t1 + n;
    }
}
```

\[(R^* \ [N] \ L^*)\]
Examples

```java
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        bal = t1 + n;
    }
}
```

The correct version:

```java
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        bal = t1 + n;
    }
}
```

The incorrect version:

```java
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        synchronized(m) {
            bal = t1 + n;
        }
    }
}
```

The correct version respects the rule of acquiring and releasing the lock only once per method call.
Dynamic Analysis for Atomicity

- **Atomizer** [POPL’04]
  - based on reduction, abstracts ops as R/L/M/N
  - leads to false alarms

- **Other techniques:** [Wang-Stoller 06], [Xu-Bodik-Hill 06], [Hatcliff et al. 04], [Park-Lu-Zhou 09]

- **Velodrome** [PLDI 08]
  - reason about serializability via happens-before relation
  - precise for observed trace, no false alarms
int x = 0;
volatile int b = 1;

**Thread A**

```java
while (true) {
    loop until b == 1;
    atomic {
        x = x + 100;
        b = 2;
    }
}
```

**Thread B**

```java
while (true) {
    loop until b == 2;
    atomic {
        x = x - 100;
        b = 1;
    }
}
```
**Execution Trace**

**Thread A**

```java
while (true) {
    loop until b == 1;
    atomic {
        x = x + 100;
        b = 2;
    }
}
```

**Thread B**

```java
while (true) {
    loop until b == 2;
    atomic {
        x = x - 100;
        b = 1;
    }
}
```
Happens-Before Ordering on Operations

- program order
Happens-Before Ordering on Operations

- program order
- synchronization order
Happens-Before Ordering on Operations

- program order
- synchronization order
- communication order
Transactional Happens-Before Ordering

\[
\text{atomic } \{
\begin{align*}
t1 &= x \\
x &= t1 + 100 \\
b &= 2
\end{align*}
\}
\]

\[
\text{test b == 1}
\]

\[
\text{test b == 1}
\]

\[
\text{atomic } \{
\begin{align*}
t1 &= x \\
x &= t1 + 100 \\
b &= 2
\end{align*}
\}
\]

\[
\text{test b == 2}
\]

\[
\text{test b == 2}
\]

\[
\text{test b == 2}
\]

\[
\text{atomic } \{
\begin{align*}
t2 &= x \\
x &= t2 - 100 \\
b &= 1
\end{align*}
\}
\]

Theorem

Transactional HB order has no cycles if and only if Trace is serializable.
Equivalent
Serial
Trace
test b == 2

atomic {
    t1 = x
    x = t1 + 100
    b = 2
}

test b == 1

atomic {
    t1 = x
    x = t1 + 100
    b = 2
}

test b == 1

atomic {
    t2 = x
    x = t2 - 100
    b = 1
}

test b == 2

atomic {
    t2 = x
    x = t2 - 100
    b = 1
}

atomic {
    t2 = x
    x = t2 - 100
    b = 1
}

atomic {
    t2 = x
    x = t2 - 100
    b = 2
}

test b == 2

atomic {
    t2 = x
    x = t2 - 100
    b = 2
}

test b == 2
Thread A

while (true) {
    loop until $b == 2$
    atomic {
        $x = x + 100$
        $b = 2$
    }
}

Thread B

while (true) {
    loop until $b == 2$
    atomic {
        $x = x - 100$
        $b = 1$
    }
}

Atomicity Violation

Cycle in transactional HB order
⇒ trace is not serializable
⇒ report atomicity violation
Controlling Thread Interference: #3 Enforce Atomicity

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

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

- Can use sequential reasoning in atomic methods
- 90% of methods
Controlling Thread Interference: #3 Enforce Atomicity

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

- 10% of methods
- No information about thread interference
- Local atomic blocks awkward

```java
void busy_wait() {
    acq(m);
    thread interference?
    while (!test()) {
        thread interference?
        rel(m);
        thread interference?
        acq(m);
        thread interference?
        x++;
        thread interference?
    }
}
```
Controlling Thread Interference: #3 Enforce Atomicity

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

```c
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```

```c
void busy_wait() {
    acq(m);
    while (!test()) {
        rel(m);
        acq(m);
        x++;
    }
}
```

Bimodal Semantics
atomic vs. read-modify-write
Controlling Thread Interference: #4 Cooperative Multitasking

- Cooperative scheduler performs context switches only at yield statements
- **Clean semantics**
  - Sequential reasoning valid by default ...
  - ... except where yields highlight thread interference
- Limitation: Uses only a single processor
Yield-Oriented Concurrency

Cooperative Scheduler
- Sequential Reasoning
- Except at yields

Preemptive Scheduler
- Full performance
- No overhead

Yield Correctness

acq(m) x = 0
rel(m) yield

... barrier yield
... yield

acq(m) x = 2
rel(m) yield

Cooperative Correctness

Yields mark all interference points

Preemptive Correctness

Yield-Oriented Concurrency

∧

Wrong
Benefits of Yield over Atomic

- Atomic methods are those with no yields

```c
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```

```c
void busy_wait() {
    acq(m);
    while (!test()) {
        rel(m);
        acq(m);
        x++;
    }
}
```
**Benefits of Yield over Atomic**

- Atomic methods are those with no yields

```c
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```

```c
atomic void busy_wait() {
    acq(m);
    while (!test()) {
        rel(m);
        yield;
        acq(m);
        x++;
    }
}
```

- **atomic** is a method-level spec.
- **yield** is a code-level spec.
Benefits of Yield over Atomic

```
atomic void copy() {
    x = 0;
    while (x < len) {
        tmp = a[x];
        b[x] = tmp;
        x++;
    }
}
```

```
void busy_wait() {
    acq(m);
    while (!test()) {
        rel(m);
        yield;
        acq(m);
        x++;
    }
}
```

*x++ is always an increment operation*
## 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>

### Non-Interference Specification
- **atomic**
- **yield**
- **yield-oriented programming**
- **automatic mutual exclusion**

---

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

\texttt{x++}

is a non-atomic read-modify-write

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

Single Thread

\texttt{x++}

\begin{verbatim}
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
\end{verbatim}
Single Thread

\[ x++ \]

Yield-Oriented Programming

\[ x++ \ vs. \ yield; \]

\begin{verbatim}
x = 0;
while (x < len) {
    yield;
    tmp = a[x];
    yield;
    b[x] = tmp;
    x++; 
}
\end{verbatim}

\begin{verbatim}
x = 0;
while (x < len) {
    yield;
    tmp = a[x];
    yield;
    b[x] = tmp;
    x++; 
}
\end{verbatim}
Yield-Oriented Programming Examples

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;
    }

    synchronized void getChars(int, int, char[], int) {...} 

    synchronized void expandCapacity(int) {...} 

    synchronized int length() {...}
}
volatile int x;

void update_x() {
    x = slow_f(x);
}

No yield between accesses to x
void update_x() {
    acquire(m);
    x = slow_f(x);
    release(m);
}

But...
Bad performance

Cooperative Correctness \land \text{Coop/preemptive Equivalence} \implies \text{Preemptive Correctness}
void update_x() {
    int fx = slow_f(x);
    acquire(m);
    x = fx;
    release(m);
}

Cooperative Correctness \land Coop/preemptive Equivalence \implies Preemptive Correctness

No yield between accesses to x
void update_x() {
    int fx = slow_f(x);
    yield;
    acquire(m);
    x = fx;
    release(m);
}

Stale value after yield

Cooperative Correctness ∧ Coop/preemptive Equivalence → Preemptive Correctness
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);

        if (x == y) {
            y = fy; return;
        }
        y = x;
    }
}
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);
        acquire(m);
        if (x == y) {
            x = fy; release(m); return;
        }
        y = x;
        release(m);
    }
}
Do Yields Help?

• Hypothesis: Yields help code comprehension and defect detection

• User study [Sadowski, Yi PLATEAU 2010]

• Methodology
  – Web-based survey, background check on threads
  – Two groups: shown code with or without yields
  – Three code samples, based on real-world bugs
  – Task: Identify all bugs
## Do Yields Help?

<table>
<thead>
<tr>
<th>StringBuffer</th>
<th>Concurrency bug</th>
<th>Some other bug</th>
<th>Didn’t find bug</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yields</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>No Yields</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All Samples</th>
<th>Concurrency bug</th>
<th>Some other bug</th>
<th>Didn’t find bug</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yields</td>
<td>30</td>
<td>3</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>No Yields</td>
<td>17</td>
<td>6</td>
<td>21</td>
<td>44</td>
</tr>
</tbody>
</table>

Difference is statistically significant
Static Program Analysis for Cooperability
**JCC: Cooperability Checker for Java**

- **Extension of Java's type system**
- **Input:** Java code with
  - traditional synchronization
  - yield annotations
  - annotations on racy variables (verified separately)
- **Theorem:** Well-typed programs are yield correct (cooperative-preemptive equivalent)

```java
class A {
    int x;  //@racy
    void m() {
        ...
        yield;
        synchronized...
        ...
    }
}
```
Identifying Yield-Correct Code

• Commuting Classifications

  R  Right-mover       Acquire
  L  Left-mover        Release
  M  Both-mover        Race-Free Access
  N  Non-mover         Racy Access
  Y  Yielding          yield

• Cooperable blocks have the pattern:

  \(((R^* [N] L^*) Y)^* [R^* [N] L^*]\)
Coop/Preemptive Equivalence

• Trace is coop/preemptive equivalent if each thread satisfies DFA
Examples

```java
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
    }
    yield;
    synchronized(m) {
        bal = t1 + n;
    }
}
```

```
acquire(m)
t1 = bal
release(m)
yield...
acquire(m)
bal = t1 + n
release(m)
```

```
((R* [N] L*) Y)* [R* [N] L*]
```
### Traces

#### Preemptive

- `acquire(m)`
- ...
- ...
- `t1 = bal`
- `release(m)`
- ...
- `yield`
- ...
- `yield`
- `acquire(m)`
- `bal = t1 + n`
- ...
- `yield`
- `release(m)`
- `yield`

#### Cooperative

- `acquire(m)`
- `t1 = bal`
- `release(m)`
- `yield`
- ...
- ...
- `yield`
- `acquire(m)`
- `bal = t1 + n`
- `release(m)`
- `yield`
- `yield`
Summary of Static Analysis

• Compute an effect for each program expression/statement
• Effect summarizes how that computation interacts with other threads

<table>
<thead>
<tr>
<th>Letter</th>
<th>Definition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Right-mover</td>
<td>Acquire</td>
</tr>
<tr>
<td>L</td>
<td>Left-mover</td>
<td>Release</td>
</tr>
<tr>
<td>M</td>
<td>Both-mover</td>
<td>Race-Free Access</td>
</tr>
<tr>
<td>N</td>
<td>Non-mover</td>
<td>Racy Access</td>
</tr>
<tr>
<td>Y</td>
<td>Yielding</td>
<td>yield</td>
</tr>
</tbody>
</table>
class TSP {

    volatile int shortestPathLength;

    void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength) {
                    shortestPathLength = path.length();
                }
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}

((R* [N] L*) Y)* [R* [N] L*]
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
```java
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            if (path.length() < shortestPathLength)
                shortestPathLength = path.length();
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
```
class StringBuffer {
    int count;

    non-mover
    synchronized int length() {
        return count;
    }
}

StringBuffer sb;

synchronized (sb) {
    if (sb.length() < 10)
        sb.add("moo");
}
class StringBuffer {
    int count;

    this ? mover : non-mover
    synchronized int length() {
        return count;
    }
}

StringBuffer sb;
synchronized (sb) {
    if (sb.length() < 10)
        sb.add("moo");
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {
        yield;
        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {
            yield;
            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {
                yield;
                searchFrom(c);
            }
        }
    }
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    compound void searchFrom(Path path) {

        if (path.length() >= shortestPathLength) return;

        if (path.isComplete()) {

            synchronized(lock) {
                if (path.length() < shortestPathLength)
                    shortestPathLength = path.length();
            }
        } else {
            for (Path c : path.children()) {

                searchFrom(c);
            }
        }
    }
}
Full Effect Lattice

one transaction that does not commute

series of transactions that do not commute

one transaction that commutes with other thread operations
<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LOC)</th>
<th>Annotation Time (min.)</th>
<th>Annotation Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.zip.Inflater</td>
<td>317</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>java.util.zip.Deflater</td>
<td>381</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>java.lang.StringBuffer</td>
<td>1,276</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>2,307</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>java.io.PrintWriter</td>
<td>534</td>
<td>40</td>
<td>109</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>1,019</td>
<td>25</td>
<td>43</td>
</tr>
<tr>
<td>java.util.zip.ZipFile</td>
<td>490</td>
<td>30</td>
<td>62</td>
</tr>
<tr>
<td>sparse</td>
<td>868</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>tsp</td>
<td>706</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>elevator</td>
<td>1,447</td>
<td>30</td>
<td>64</td>
</tr>
<tr>
<td>raytracer-fixed</td>
<td>1,915</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>sor-fixed</td>
<td>958</td>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>1,352</td>
<td>10</td>
<td>39</td>
</tr>
<tr>
<td>Total</td>
<td>13,570</td>
<td>231</td>
<td>490</td>
</tr>
<tr>
<td>Total per KLOC</td>
<td></td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>
## Number of Interference Points

<table>
<thead>
<tr>
<th>Program</th>
<th>No Spec</th>
<th>Race</th>
<th>Atomic</th>
<th>Atomic Race</th>
<th>Yield</th>
<th>Unintended Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.zip.Inflater</td>
<td>36</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>java.util.zip.Deflater</td>
<td>49</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>java.lang.StringBuffer</td>
<td>210</td>
<td>81</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>230</td>
<td>87</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>java.io.PrintWriter</td>
<td>73</td>
<td>99</td>
<td>130</td>
<td>97</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>185</td>
<td>106</td>
<td>44</td>
<td>24</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>java.util.zip.ZipFile</td>
<td>120</td>
<td>105</td>
<td>85</td>
<td>53</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>sparse</td>
<td>329</td>
<td>98</td>
<td>48</td>
<td>14</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>tsp</td>
<td>445</td>
<td>115</td>
<td>437</td>
<td>80</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>elevator</td>
<td>454</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raytracer-fixed</td>
<td>565</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sor-fixed</td>
<td>249</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>983</td>
<td>130</td>
<td>47</td>
<td>37</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

### Interference at:
- All field accesses
- All lock acquires
- Atomic method calls

### Fewer Interference Points: Easier to Reason about Code!
Dynamic Program Analysis for Cooperability
yield;
acquire(m);
while(x>0) {
    release(m);
    acquire(m);
}
assert x==0;
release(m);
yield;

Copper
[PPOPP 11]
Copper

- Build Transactional Happens-Before
  - program order
  - sync. order
  - comm. order

yield
acquire(m)
test x > 0
release(m)
yield
acquire(m)
test x > 0
release(m)
...
yield
acquire(m)
x = 1
release(m)
yield
Copper

- **Build Transactional Happens-Before**
- **Yields mark transaction ends**
- **Cycles indicate missing yields**

```
yield
acquire(m)
test x > 0
release(m)
yield
```

```
acquire(m)
x = 1
release(m)
yield
```
Copper

```c
yield;
acquire(m);
while(x>0) {
    release(m);
    yield;
    acquire(m);
}
assert x==0;
release(m);
yield;
```
## Copper Results

<table>
<thead>
<tr>
<th>program</th>
<th>LLOC</th>
<th>No Analysis</th>
<th>Atomic Methods</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse</td>
<td>712</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sor</td>
<td>721</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>series</td>
<td>811</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>1083</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>1299</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>elevator</td>
<td>1447</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lufact</td>
<td>1472</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>raytracer</td>
<td>1862</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3557</td>
<td>377</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>hedc</td>
<td>6409</td>
<td>305</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>mtrt</td>
<td>6460</td>
<td>695</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>raja</td>
<td>6863</td>
<td>396</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>colt</td>
<td>25644</td>
<td>601</td>
<td>113</td>
<td>13</td>
</tr>
<tr>
<td>jigsaw</td>
<td>48674</td>
<td>3415</td>
<td>550</td>
<td>47</td>
</tr>
</tbody>
</table>

Interference at:
- field accesses
- all lock acquires
- atomic method calls

in non-atomic methods

Fewer interference points: less to reason about!
Cooperative Scheduler
- Sequential Reasoning
- Except at yields

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

... barrier
yield

... yield

acq(m)
\[x = 2\]
rel(m)
yield

Yield Correctness

Cooperative Correctness ∨ Preemptive Correctness

Preemptive Scheduler
- Full performance
- No overhead

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

... barrier
yield

... yield

acq(m)
\[x = 2\]
rel(m)
yield

Yields mark all thread interference
Summary

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

• Atomicity
  – code behaves as if atomic methods executed serially
  – [http://users.soe.ucsc.edu/~cormac/atom.html](http://users.soe.ucsc.edu/~cormac/atom.html)

• Yield-oriented programming
  – code behaves as if run on cooperative scheduler
  – sequential reasoning ok, except where yields document thread interference (1-10/KLOC)
  – [http://users.soe.ucsc.edu/~cormac/coop.html](http://users.soe.ucsc.edu/~cormac/coop.html)
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
Where To Go From Here?

- Other analyses for yield correctness
- Other non-interference properties
  - determinism, ...
- Deterministic schedulers
- Record-and-replay
- Other programming models
  - domain-specific
  - multicore and distributed programming
Key References

Key References