Static and Dynamic Analyses for Reliable Concurrency

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)
- 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
Multicore CPUs
Concurrent Programming Models

- Multiple threads, shared memory, sync

**Multithreaded programming is difficult.**
- *schedule-dependent* behavior
- race conditions, deadlocks, atomicity violations, ...
- difficult to detect, reproduce, or eliminate
Multiple Threads

\texttt{x++}

is a non-atomic
read-modify-write

\begin{verbatim}
x = 0;
thread interference?
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}
Controlling Thread Interference: #1 Manually

1 Inspect code
2 Identify where interference does not occur
Controlling Thread Interference: 
#1 Manually w/ Productivity Heuristic

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

1. Assume no interference
2. Use sequential reasoning

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

- 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
Thread B
  t2 = bal
  bal = t2 - 10
```
Controlling Thread Interference: #2 Enforce Race Freedom

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

Thread A

\[
\begin{align*}
  \ldots & \\
  t1 &= bal; \\
  bal &= t1 + 10; \\
  \ldots & 
\end{align*}
\]

Thread B

\[
\begin{align*}
  \ldots & \\
  t2 &= bal; \\
  bal &= t2 - 10; \\
  \ldots & 
\end{align*}
\]

Thread A

\[
\begin{align*}
  t1 &= bal \\
  t2 &= bal \\
  bal &= t1 + 10 \\
  bal &= t2 - 10
\end{align*}
\]
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
- Static and dynamic analysis tools to detect races

- 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 = 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:
#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

```plaintext
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 are atomic
**Bohr: Static Analysis for Atomicity**

- Extension of Java's type system [TOPLAS’08]
- Input: Java code with
  - traditional synchronization
  - 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]

Serializable 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;
    }
}
```

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

```
acquire(m)
...
acquire(m)
...
acquire(m)
...
```

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

(R* [N] L*)
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 1
while (true) {
    loop until b == 1;
    atomic {
        x = x + 100;
        b = 2;
    }
}

Thread 2
while (true) {
    loop until b == 2;
    atomic {
        x = x - 100;
        b = 1;
    }
}
### Execution Trace

#### Thread 1

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

#### Thread 2

```plaintext
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
Transaction Happens-Before Ordering

**Theorem**

Transactional HB order has no cycles if and only if Trace is serializable.
Equivalent Serial Trace
Equivalent Serial Trace

\[
\text{atomic} \quad \{
qquad t_1 = x \\
qquad x = t_1 + 100 \\
qquad b = 2
\}
\]

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

\[
\text{atomic} \quad \{
qquad t_2 = x \\
qquad x = t_2 - 100 \\
qquad b = 1
\}
\]

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

\[
\text{atomic} \quad \{
qquad t_1 = x \\
qquad x = t_1 + 100 \\
qquad b = 2
\}
\]

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

\[
\text{atomic} \quad \{
qquad t_2 = x \\
qquad x = t_2 - 100 \\
qquad b = 1
\}
\]

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

\[
\text{atomic} \quad \{
qquad t_2 = x \\
qquad x = t_2 - 100 \\
qquad b = 1
\}
\]

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

Thread 1

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

Thread 2

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

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

```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 are atomic
- Static and dynamic analyses
Controlling Thread Interference: #3 Enforce Atomicity

- 10% of methods not atomic
- Local atomic blocks awkward
- Atomicity provides no information about thread interference

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

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

Bimodal Semantics
increment vs. non-atomic
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 Programming

Cooperative Scheduler
- Sequential Reasoning
- Except at yields

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

...  
barrier
yield

...  
yield

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

Preemptive Scheduler
- Full performance
- No overhead

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

...  
barrier
yield

...  
yield

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

Yield Correctness:  
yields mark all thread interference

Cooperative Correctness  \(\wedge\)  Preemptive Correctness
Yield vs. 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++;
    }
}
```
Yield vs. Atomic

- Atomic methods are those with no yields

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

- `atomic` is a method-level spec.
- `yield` is a code-level spec.

```c
void busy_wait() {
    acq(m);
    while (!test()) {
        rel(m);
        yield;
        acq(m);
        x++;
    }
}
```
## Non-Interference Design Space

<table>
<thead>
<tr>
<th>Policy Enforcement</th>
<th>Non-Interference Specification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>traditional sync + analysis</td>
<td>atomic</td>
<td>yield</td>
</tr>
<tr>
<td></td>
<td>atomicity, serializability</td>
<td>Yield-oriented programming</td>
</tr>
<tr>
<td>new run-time systems</td>
<td>transactional memory</td>
<td>automatic mutual exclusion</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

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

Single Thread

```
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
```
Yield-Oriented Programming

\[
x++ \quad \text{vs.} \quad \text{yield};
\]

Single Thread

\[
x++
\]

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

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

Cooperative Correctness \land \text{Yield Correctness} \implies \text{Preemptive Correctness}

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

But...
Bad performance

Version 2

Cooperative Correctness \land Yield Correctness \rightarrow Preemptive Correctness
```c
void update_x() {
    int fx = slow_f(x);
    acquire(m);
    x = fx;
    release(m);
}
```

**Version 3**

Cooperative Correctness \(\land\) Yield Correctness → Preemptive Correctness

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

---

Version 4

- Stale value after yield

Cooperative Correctness $\land$ Yield Correctness $\Rightarrow$ Preemptive Correctness
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);
        if (x == y) {
            x = 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 Yield Correctness
**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)
Identifying Serializable Code

• Compute an effect for each stmt to summarize how stmt interacts with other threads

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

• Serializable blocks have the pattern: 
  \[ R^* [N] L^* \]
Identifying Yield-Correct Code

- Compute an effect for each stmt to summarize how stmt interacts with other threads

  R  Right-mover
  L  Left-mover
  M  Both-mover
  N  Non-mover
  Y  Yielding

- Yield-correct threads have the pattern:
  \(((R^* [N] L^*) Y)^* (R^* [N] L^*)\)
DFA for Yield-Correctness

- Trace is yield-correct if each thread satisfies DFA
Examples

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

<table>
<thead>
<tr>
<th>acquire(m)</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>t1 = bal</td>
<td>release(m)</td>
</tr>
<tr>
<td>release(m)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
<tr>
<td>acquire(m)</td>
<td>bal = t1 + n</td>
</tr>
<tr>
<td>bal = t1 + n</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
<tr>
<td>release(m)</td>
<td>...</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
</tbody>
</table>

### Cooperative

<table>
<thead>
<tr>
<th>acquire(m)</th>
<th>t1 = bal</th>
</tr>
</thead>
<tbody>
<tr>
<td>release(m)</td>
<td>yield</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>acquire(m)</td>
<td>bal = t1 + n</td>
</tr>
<tr>
<td>bal = t1 + n</td>
<td>release(m)</td>
</tr>
<tr>
<td>...</td>
<td>yield</td>
</tr>
<tr>
<td>yield</td>
<td>...</td>
</tr>
<tr>
<td>release(m)</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 Path {
    ... // methods like mover int length() and mover boolean isComplete()
    ...
}

one transaction that commutes with other thread operations
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);
            }
        }
    }
}

((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;
        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) {
        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);
      }
    }
  }
}
class StringBuffer {
    int count;

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

    non-mover
    synchronized void add(String s) {
        ...
    }
}

StringBuffer sb;

synchronized (sb) {
    if (sb.length() < 10)
        sb.add("moo");
}
Conditional Effects

class StringBuffer {
  int count;

  synchronized int length() {
    return count;
  }

  synchronized void add(String s) {
    ...
  }
}

StringBuffer sb;

synchronized (sb) {
  if (sb.length() < 10)
    sb.add("moo");
}
Full Effect Lattice

one transaction that does not commute

one transaction that commutes with other thread operations

series of transactions that do not commute
<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LOC)</th>
<th>Annotation Time (min.)</th>
<th>Anotation 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><strong>Total</strong></td>
<td><strong>13,570</strong></td>
<td><strong>231</strong></td>
<td><strong>490</strong></td>
</tr>
<tr>
<td><strong>Total per KLOC</strong></td>
<td></td>
<td></td>
<td><strong>36</strong></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>0</td>
</tr>
<tr>
<td>raytracer-fixed</td>
<td>565</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>sor-fixed</td>
<td>249</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>983</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>3,928</td>
<td>1,291</td>
<td>1,890</td>
<td>432</td>
<td>180</td>
<td>13</td>
</tr>
<tr>
<td><strong>Total per KLOC</strong></td>
<td>289</td>
<td>95</td>
<td>139</td>
<td>32</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>

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

*in non-atomic methods*

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

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

• Build Transactional Happens-Before
  – program order
  – sync. order
  – comm. order
Copper

- Build Transactional Happens-Before

- Yields mark transaction ends

- Cycles indicate missing yields
yield;
acquire(m);
while(x>0) {
    release(m);
    yield;
    acquire(m);
}
assert x==0;
release(m);
yield;

acquire(m)
read x
release(m)
yield

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

acquire(m)
read x
release(m)
...

yield
RoadRunner Framework for Dynamic Concurrency Analyses
[PASTE ‘10, github]

Error: ...

Java
Bytecode

Others: Sofya [KDR 07], CalFuzzer [JNPS 09]
## 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>49</td>
<td>0</td>
</tr>
<tr>
<td>sort</td>
<td>721</td>
<td>3</td>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>server</td>
<td>811</td>
<td>0</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>crypt</td>
<td>1083</td>
<td>0</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>moldyn</td>
<td>1299</td>
<td>0</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>elevator</td>
<td>1447</td>
<td>3</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>lufact</td>
<td>1472</td>
<td>3</td>
<td>57</td>
<td>3</td>
</tr>
<tr>
<td>raytracer</td>
<td>1862</td>
<td>3</td>
<td>65</td>
<td>3</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3557</td>
<td>1</td>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>hedc</td>
<td>6409</td>
<td>2</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>mtrt</td>
<td>6460</td>
<td>1</td>
<td>65</td>
<td>2</td>
</tr>
<tr>
<td>raja</td>
<td>6863</td>
<td>0</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>colt</td>
<td>25644</td>
<td>13</td>
<td>113</td>
<td>13</td>
</tr>
<tr>
<td>jigsaw</td>
<td>48674</td>
<td>47</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!
Yield-oriented Programming

Cooperative Scheduler
- Sequential Reasoning
- Except at yields

Preemptive Scheduler
- Full performance
- No overhead

Yield Correctness:
yields mark all thread interference

Cooperative Correctness

Preemptive Correctness
Summary

• Race freedom
  - code behaves \textit{as if} on sequentially consistent machine

• Atomicity
  - code behaves \textit{as if} atomic methods executed serially

• Yield-oriented programming
  - code behaves \textit{as if} run on cooperative scheduler
  - sequential reasoning ok, except at yields \((1-10/\text{KLOC})\)
  - \url{http://users.soe.ucsc.edu/~cormac/coop.html}

• Other analyses for yield correctness
• Other non-interference properties: determinism, ...
• Deterministic schedulers, record-and-replay
• Other programming models/hardware platforms
Summary

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

• Atomicity
  – code behaves as if atomic methods executed serially

• Yield-oriented programming
  – use traditional synchronization & multicore hardware
  – document all interference with yields
  – static analyses check interference only at yields
  – code behaves as if run on cooperative scheduler
  – sequential reasoning ok, except at yields (1-10/KLOC)
  – http://users.soe.ucsc.edu/~cormac/coop.html
Summary

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

• Atomicity
  – code behaves as if atomic methods executed serially

• 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)
Future Directions

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