Types for Precise Thread Interference

Jaeheon Yi, Tim Disney, Cormac Flanagan
UC Santa Cruz

Stephen Freund
Williams College

October 23, 2011
Multiple Threads

\[
x++
\]
is a non-atomic read-modify-write

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

Single Thread

\[
x++
\]

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

Programmer Productivity Heuristic:
assume no interference, use sequential reasoning
Controlling Thread Interference #2: Race Freedom

- Race condition: two concurrent unsynchronized accesses, at least one write
- Strongly correlated with defects
- Race-free programs exhibit *sequentially consistent* behavior, even when run on a relaxed memory model
- Race freedom by itself is not sufficient to prevent concurrency bugs
Controlling Thread Interference #3: Atomicity

- A method is **atomic** if it behaves *as if* it executes serially, without interleaved operations of other thread

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

```c
void busyloop(...) {
    acquire(m);
    while (!test()) {
        release(m);
        acquire(m);
        x++;
    }
}
```

sequential reasoning ok
90% of methods atomic

10% of methods non-atomic
local atomic blocks awkward
full complexity of threading

**bimodal semantics**
increment or read-modify-write
Review of 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
Cooperative Concurrency

Code with sync & yields
... acquire(m) x++ release(m)
yield // interference ...

Cooperative scheduler
seq. reasoning ok except where yields highlight interference

Preemptive scheduler
full performance no overhead

Yields mark all thread interference

Cooperative correctness
\^ Coop/preemptive equivalence \implies Preemptive correctness
Benefits of Yield over Atomic

• Atomic methods are exactly those with no yields

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

```c
void busyloop(...) {
    acquire(m);
    while (!test()) {
        release(m);
        yield;
        acquire(m);
        x++;
    }
}
```

atomic is an interface-level spec (method contains no yields)  
yield is a code-level spec
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

\[ x++ \]

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

x++ is an increment

{ int t=x; yield; x=t+1; }

Single Thread

x++

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

x = 0;
while (x < len) {
tmp = a[x];
b[x] = tmp;
x++;
}
Cooperability in the design space

<table>
<thead>
<tr>
<th>Policy</th>
<th>Non-interference specification</th>
<th>Atomic</th>
<th>Yield</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional synchronization + analysis</td>
<td></td>
<td>atomicity</td>
<td>cooperability</td>
</tr>
<tr>
<td>New runtime systems</td>
<td></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
Cooperative Concurrency

1. Static type system for verifying C-P equivalence

2. Example of coding with Yields

3. Experiments: Yield count
Type System for Cooperative-Preemptive Equivalence

- Type checker takes as input Java programs with
  - traditional synchronization
  - yield annotations
  - racy variables (if any) are identified
    - (other type systems/analyses identify races)

- Theorem:
  
  Well-typed programs are cooperative-preemptive equivalent
Effect Language

• Approach: Compute an effect for each program expression that summarizes how that expression interacts with other threads

• Effects:
  • R    right-mover        lock acquire
  • L    left-mover           lock release
  • M    both-mover        race-free access
  • N    non-mover         racy access
  • Y    yield

• Lipton’s theory of reduction: Code block is serializable if matches \( R^* [N] L^* \)

• Program is cooperative-preemptive equivalent
  • if each thread matches: \( (R^* [N] L^* Y)^* (R^* [N] L^*) \)
  • (serializable transactions separated by yields)
Sequential Effect Composition

<table>
<thead>
<tr>
<th>;</th>
<th>F</th>
<th>Y</th>
<th>M</th>
<th>R</th>
<th>L</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>Y</td>
<td>M</td>
<td>R</td>
<td>L</td>
<td>N</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>Y</td>
<td>M</td>
<td>R</td>
<td>L</td>
<td>N</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>Y</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>—</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>R</td>
<td>N</td>
<td>—</td>
<td>N</td>
<td>—</td>
</tr>
</tbody>
</table>

\( t=x; \)   \( x=t+1; \)

\[
\begin{array}{ccc}
N & N & \text{Error} \\
\end{array}
\]

\( t=x; \) \( \text{yield}; \) \( x=t+1; \)

\[
\begin{array}{ccc}
N & y & N \\
R & N & N \\
N &  & \\
\end{array}
\]

x is racy...
Effect Composition for Choice

\[
\text{if (b) then acq(M); else rel(N);}
\]

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Join correctly over-approximates the effect of each branch
Full Effect tracks both commutativity and atomicity

mover effect \{F, Y, M, R, L, N\} combined with serializability effect \{A, C\}
Effect Language includes Conditional Effects

- Constants: AF, AM, AL, AR, AN, CY, CM, CR, CL, CN

- Conditionals encode effects under different locking conditions:
  \[ \text{lock} \ ? \ \text{effect} : \text{effect} \]

```java
class StringBuffer {
    int count;

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

    ...
}
```
Cooperative Concurrency

1. Static type system for verifying C-P equivalence
   Cooperative correctness

2. Example of coding with Yields
   Code with sync & yields
   Preemptive scheduler
   full performance
   no overhead
   Cooperative scheduler
   seq. reasoning ok
   except where yields
   highlight interference

3. Experiments: Yield count
   acquire(m) x=0 release(m) yield
   barrier(b) yield
   acquire(m) x++ release(m) yield
   ... yield
   acquire(m) x=0 release(m) yield
void update_x() {
    x = slow_f(x);
}

x is volatile
concurrent calls to update_x

Not C-P equivalent:
No yield between accesses to x

version 1

Coop/preemptive equivalence \(\land\) Cooperative correctness \(\Rightarrow\) Preemptive correctness
void update_x() {
    synchronized(m) {
        x = slow_f(x);
    }
}

Not efficient!
high lock contention
= low performance

version 2
Coop/preemptive equivalence \(\land\)
Cooperative correctness \(\implies\)
Preemptive correctness
void update_x() {
    int fx = slow_f(x);

    synchronized(m){
        x = fx;
    }
}
void update_x() {
    int fx = slow_f(x);
    yield;
    synchronized(m){
        x = fx;
    }
}

Not correct:
Stale value at yield

version 4
Coop/preemptive equivalence ∧ Cooperative correctness ⇒ Preemptive correctness
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);

        if (x == y) {
            x = fy;
            return;
        } else {
            y = x;
        }
    }
}

restructure:
test and retry pattern

version 5  
Coop/preemptive equivalence ∧ Cooperative correctness  ⇒  Preemptive correctness

Not C-P equivalent:  
No yield between access to x
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);
        synchronized(m) {
            if (x == y) {
                x = fy;
                return;
            } else {
                y = x;
            }
        }
    }
}

version 6

Coop/preemptive equivalence \(\land\) Cooperative correctness \(\Rightarrow\) Preemptive correctness
Cooperative Concurrency

2. Examples of coding with Yields

Cooperative scheduler
seq. reasoning ok except where yields highlight interference

Preemptive scheduler
full performance no overhead

3. Experiments: Yield count

1. Static type system for verifying C-P equivalence

Cooperative correctness

Cooperative/P preemptive equivalence

Cooperative/P correctness
### Interference Points: by the numbers

In non-atomic methods, count field accesses, lock acquires, and atomic methods calls

<table>
<thead>
<tr>
<th>program</th>
<th>LOC</th>
<th>No Analysis</th>
<th>Method Atomic</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>j.u.zInflater</td>
<td>317</td>
<td>36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j.u.zDeflater</td>
<td>381</td>
<td>49</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j.l.StringBuffer</td>
<td>1276</td>
<td>210</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>j.l.String</td>
<td>2307</td>
<td>230</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>j.i.PrintWriter</td>
<td>534</td>
<td>73</td>
<td>130</td>
<td>26</td>
</tr>
<tr>
<td>j.u.Vector</td>
<td>1019</td>
<td>185</td>
<td>44</td>
<td>4</td>
</tr>
<tr>
<td>j.u.ZipFile</td>
<td>490</td>
<td>120</td>
<td>85</td>
<td>30</td>
</tr>
<tr>
<td>sparse</td>
<td>868</td>
<td>329</td>
<td>48</td>
<td>6</td>
</tr>
<tr>
<td>tsp</td>
<td>706</td>
<td>445</td>
<td>487</td>
<td>16</td>
</tr>
<tr>
<td>elevator</td>
<td>1447</td>
<td>454</td>
<td></td>
<td></td>
</tr>
<tr>
<td>raytracer-fixed</td>
<td>1915</td>
<td>565</td>
<td>128</td>
<td>12</td>
</tr>
<tr>
<td>sor-fixed</td>
<td>958</td>
<td>249</td>
<td>128</td>
<td>12</td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>1352</td>
<td>983</td>
<td>657</td>
<td>30</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>13570</td>
<td>3928</td>
<td>1890</td>
<td>180</td>
</tr>
</tbody>
</table>
Summary of Cooperative Concurrency

Cooperative scheduler
seq. reasoning ok...
...except where yields
highlight interference
x++ an increment op

Cooperative correctness

Preemptive scheduler
full performance
no overhead

Preemptive correctness

Yields mark all thread interference

Coop/preemptive equivalence

⇒

We have:

 acquire(m)
 x++
 release(m)
 yield   // interference
...