Cooperative Concurrency for a Multicore World

Stephen Freund
Williams College

Cormac Flanagan, Tim Disney,
Caitlin Sadowski, Jaeheon Yi,
UC Santa Cruz
Controlling Thread Interference: 
#1 Manually

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

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

+ Works some of the time

× Easy to make mistakes...
Controlling Thread Interference: #2 Enforce Race Freedom

- Race: concurrent conflicting accesses

Thread A

... 
\[ t_1 = \text{bal}; \]
\[ \text{bal} = t_1 + 10; \]
...

Thread B

... 
\[ t_2 = \text{bal}; \]
\[ \text{bal} = t_2 - 10; \]
...

Thread A

\[ t_1 = \text{bal} \]
\[ \text{bal} = t_1 + 10 \]

Thread B

\[ t_2 = \text{bal} \]
\[ \text{bal} = t_2 - 10 \]
Controlling Thread Interference: #2 Enforce Race Freedom

- Race: concurrent conflicting accesses

Thread A

...  
t1 = bal;  
bal = t1 + 10;  
...

Thread B

...  
t2 = bal;  
bal = t2 - 10;  
...
Controlling Thread Interference: #2 Enforce Race Freedom

+ Races are correlated to defects
+ Race-freedom ensures sequentially-consistent behavior
× But not sufficient...

Thread A

...  
t1 = bal;  
bal = t1 + 10;  
...

Thread B

...  
t2 = bal;  
bal = t2 - 10;  
...
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 = 0
rel(m)

acq(m)
bal = t1 + 10
rel(m)
Controlling Thread Interference: #3 Enforce Atomicity

Atomic method must behave as if it executes 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 executes serially, without interleaved operations of other threads

```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
Controlling Thread Interference: #3 Enforce Atomicity

Atomic method must behave as if it executes serially, without interleaved operations of other threads

- 10% of methods aren't

- No information about thread interference

- Local atomic blocks awkward

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

Atomic method must behave as if it executes serially, without interleaved operations of other threads

```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++;
        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

× Uses only a single processor
Cooperative Concurrency

Cooperative Scheduler
- Sequential Reasoning
- Except at yields

Preemptive Scheduler
- Full performance
- No overhead

Cooperative Correctness

Preemptive Correctness

Coop/preemptive Equivalence

yields mark all interference points
Cooperative Concurrency

Cooperative Scheduler
- Sequential Reasoning
- Except at yields

Preemptive Scheduler
- Full performance
- No overhead

Coop/preemptive Equivalence

yields mark all interference points

Cooperative Correctness

Preemptive Correctness

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

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

... barrier
yield

... yield

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

... yield

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

... yield

acq(m) x = 0
rel(m) yield
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++;
    }
}
```

- `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++;
    }
}
```
Benefits of Yield over Atomic

```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);
        yield;
        acq(m);
        x++;
    }
}
```

$x++$ is always an atomic increment operation
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);
        {
            t = x;
            yield;
            x = t + 1;
        }
    }
}
```
## Cooperability in the Design Space

<table>
<thead>
<tr>
<th>Policy Enforcement</th>
<th>Non-Interference Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>atomic</td>
</tr>
<tr>
<td>traditional sync +</td>
<td>atomicity, serializability</td>
</tr>
<tr>
<td>analysis</td>
<td></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
Cooperative Concurrency

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

Cooperative Correctness

Preemptive Correctness

Coop/preemptive Equivalence
Cooperative Concurrency

**Cooperative Scheduler**
- Sequential Reasoning
- Except at yields

```plaintext
acq(m) x = 0
rel(m) yield
... barrier yield
... yield
```

**Preemptive Scheduler**
- Full performance
- No overhead

```plaintext
acq(m) x = 0
rel(m) yield
... barrier yield
... yield
acq(m) x = 2
rel(m) yield
```

**JCC**

Input: Java source + yields + race info

**Cooperative Correctness** × **Preemptive Correctness**
volatile int x;

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

No yield between accesses to x

Cooperative Correctness \land Coop/preemptive Equivalence \implies Preemptive Correctness
void update_x() {
    synchronized(m) {
        x = slow_f(x);
    }
}

But...
Bad performance

Cooperative Correctness \land\quad \text{Coop/preemptive Equivalence} \quad \Rightarrow \quad \text{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;
    }
}
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);

        if (x == y) {
            x = fy; return;
        }
        y = x;
    }
}
```c
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);
        if (x == y) {
            yield;
            x = fy; return;
        }
        y = x;
    }
    y = x;
}
```

**Version 6**

- **Stale value after yield**
- **Cooperative Correctness** ∧ **Coop/preemptive Equivalence** → **Preemptive Correctness**
void update_x() {
    int y = x;
    for (;;) {
        yield;
        int fy = slow_f(y);
        synchronized(m) {
            if (x == y) {
                x = fy; return;
            }
            y = x;
        }
    }
}

Cooperative Correctness \land Coop/preemptive Equivalence \implies Preemptive Correctness
Identifying Cooperable Code

- Commuting Classifications [Lipton 76]

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

- Serializable blocks have pattern: $R^* \ [N] \ L^*$
- Cooperable blocks have the pattern:
  $$((R^* \ [N] \ L^*) \ Y)^* \ [R^* \ [N] \ L^*]$$
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        bal = t1 + n;
    }
}

((R* [N] L*) Y)* [R* [N] L*]
void deposit(int n) {
    synchronized(m) {
        t1 = bal;
        bal = t1 + n;
    }
}

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

acquire(m) → acquire(m)
...
...
t1 = bal
release(m)
...
...
yield
...
...
yield
acquire(m)
bal = t1 + n
...
...
yield
release(m)
yield
release(m)
...
yield
Effect Language

• Commuting Effects Lattice
  
    if (test) s1 else s2

• Atomicity Effects:
  
    Atomic or Compound

• Method specs to enable modular checking:
  
    atomic mover void m1()
    atomic non-mover void m2()
    compound non-mover void m3()
Effect Language

• Conditional Effects

class Vector {
    int count;

    this ? atomic mover : atomic non-mover
    public synchronized int length() {
        return count;
    }
}
}
class TSP {
    Object lock;
    volatile int shortestPathLength; // lock held on writes

    non-mover 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.extendToAdj()) {

                searchFrom(c);
            }
        }
    }
}
<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Interference Points</th>
<th>Unintended Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Spec</td>
<td>Race</td>
</tr>
<tr>
<td>java.util.zip.Inflater</td>
<td>36</td>
<td>12</td>
</tr>
<tr>
<td>java.util.zip.Deflater</td>
<td>49</td>
<td>13</td>
</tr>
<tr>
<td>java.lang.StringBuffer</td>
<td>210</td>
<td>81</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>230</td>
<td>87</td>
</tr>
<tr>
<td>java.io.PrintWriter</td>
<td>73</td>
<td>99</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>185</td>
<td>106</td>
</tr>
<tr>
<td>java.util.zip.ZipFile</td>
<td>120</td>
<td>105</td>
</tr>
<tr>
<td>sparse</td>
<td>98</td>
<td>48</td>
</tr>
<tr>
<td>tsp</td>
<td>445</td>
<td>115</td>
</tr>
<tr>
<td>elevator</td>
<td>454</td>
<td>130</td>
</tr>
<tr>
<td>raytracer-fixed</td>
<td>565</td>
<td></td>
</tr>
<tr>
<td>sor-fixed</td>
<td>249</td>
<td>130</td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>983</td>
<td>130</td>
</tr>
<tr>
<td>Total</td>
<td>3,928</td>
<td>1,291</td>
</tr>
<tr>
<td>Total per KLOC</td>
<td>289</td>
<td>95</td>
</tr>
</tbody>
</table>

- Interference at:
  - `racy` field accesses
  - all lock acquires
  - atomic method calls in non-atomic methods
  - yield points

- Fewer Interference Points: Easier to Reason about Code

~35 Annotations/KLOC
**Cooperability**

- Document interference with yields (10-20/KLOC)
- Tools verify cooperative-preemptive equivalence
  - Cooperative scheduling for reasoning
  - Preemptive scheduling for performance
- Next steps: user studies, hybrid checkers, yield inference