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

```
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: Manually

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

manually identify where thread interference does *not* occur

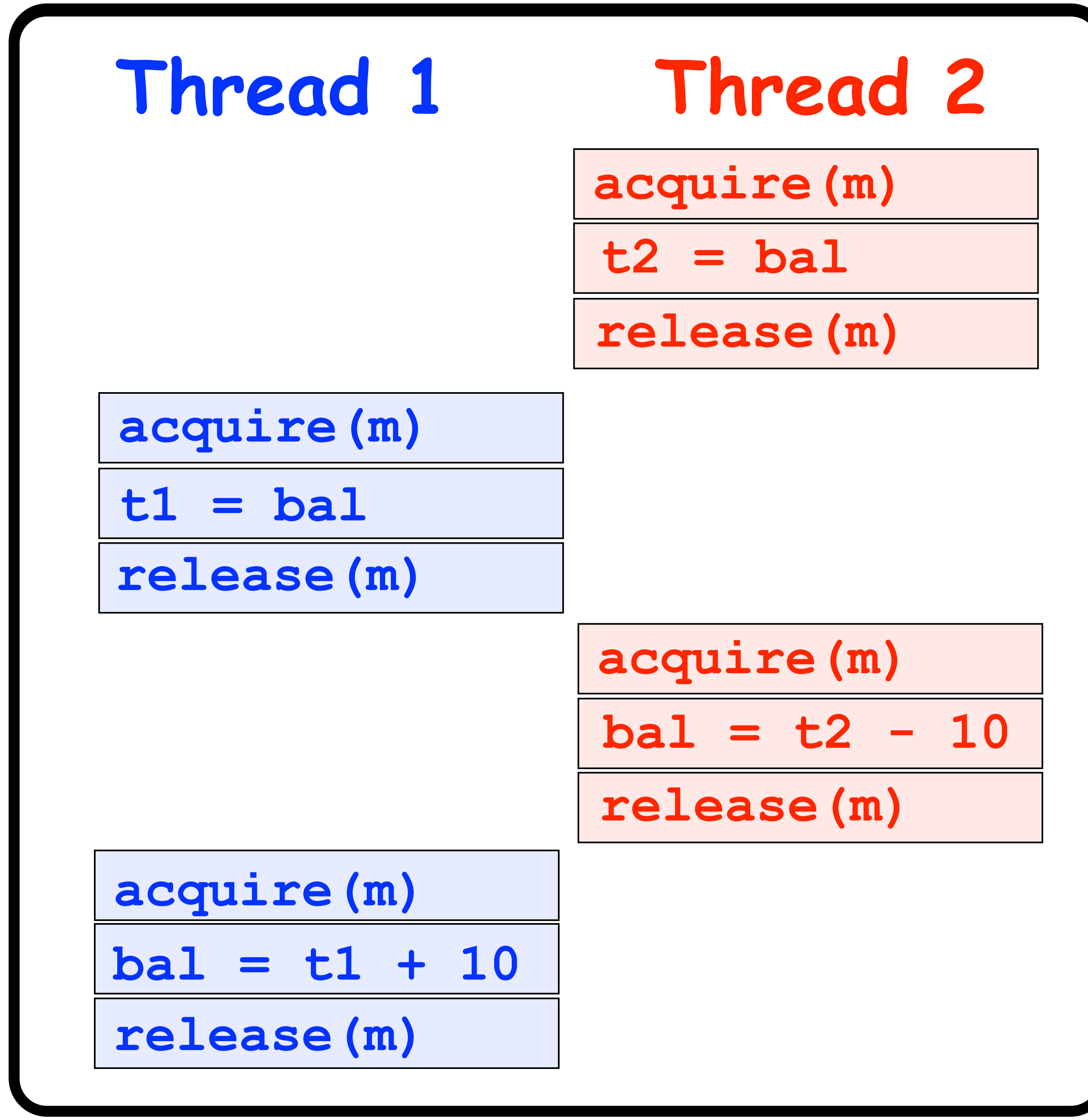
```
x = 0;

while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

}
```

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

```
atomic copy(...) {
    x = 0;

    while (x < len) {

        tmp = a[x];

        b[x] = tmp;

        x++;

    }
}
```

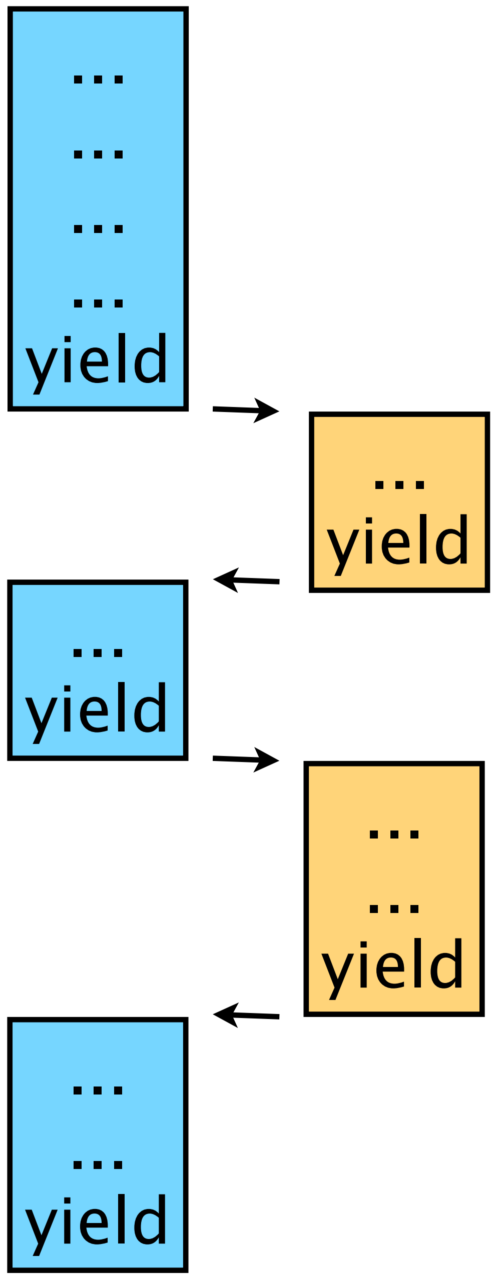
<u>bimodal semantics</u>
increment or
read-modify-write

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

sequential reasoning ok
90% of methods atomic

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

Review of Cooperative Multitasking

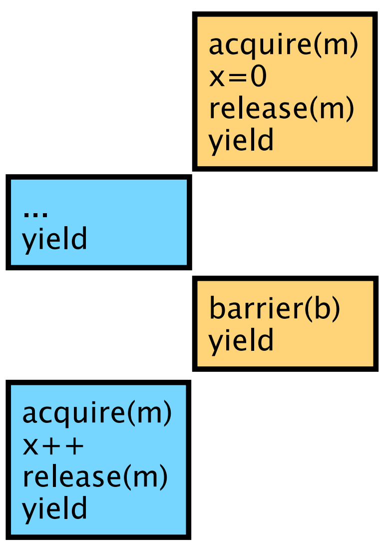


- Cooperative scheduler performs context switches only at yield statements
- Clean semantics
  - Sequential reasoning valid by default ...
  - ... except where yields highlight thread interference
- Limitation: Uses only a single processor

Cooperative Concurrency

<u>Cooperative scheduler</u>

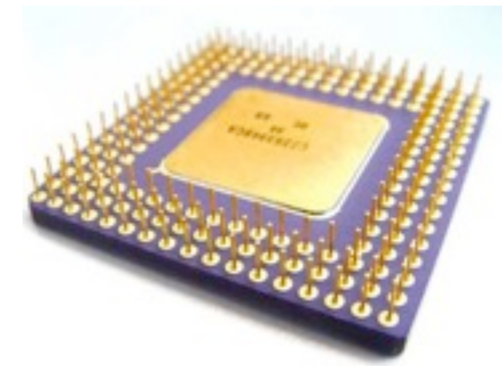
seq. reasoning ok
except where yields
highlight interference



Code with sync & yields

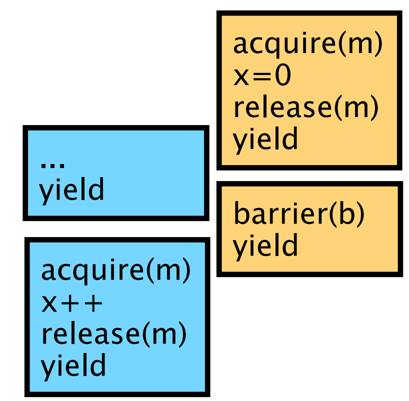
```
...
acquire(m)
x++
release(m)
yield // interference
...
```

$\cong$



<u>Preemptive scheduler</u>

full performance
no overhead



Yields mark all thread interference

Cooperative correctness	$\wedge$	Coop/preemptive equivalence	$\Rightarrow$	Preemptive correctness

Benefits of Yield over Atomic

- Atomic methods are exactly those with no yields

```
atomic copy(...) {
   x = 0;

   while (x < len) {

      tmp = a[x];

      b[x] = tmp;

      x++;

   }
}
```

x++ always an increment operation

```
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

```
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++;

}
```

Cooperative Concurrency

x++ is an increment

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

```
x = 0;

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

    x++;

}
```

Single Thread

x++

```
x = 0;

while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

}
```

Cooperability in the design space

non-interference specification

policy	`atomic`	`yield`
traditional synchronization + analysis	atomicity	cooperability (this talk)
new runtime systems	transactional memory	automatic mutual exclusion

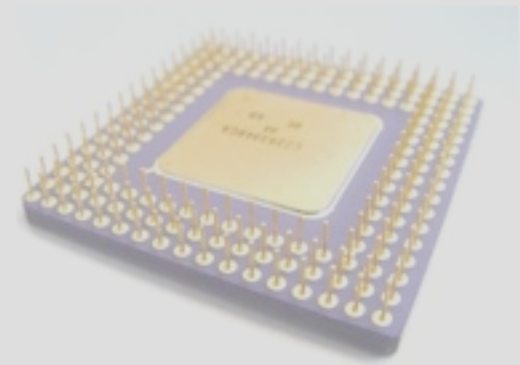
Transactional Memory, Larus & Rajwar, 2007
Automatic mutual exclusion, Isard & Birrell, HOTOS '07

Cooperative Concurrency

Cooperative scheduler
seq. reasoning ok
except where yields
highlight interference

Code with sync & yields
yield
...

2. Example of coding with Yields



Preemptive scheduler
full performance
no overhead

acquire(m)
x=0
release(m)

3. Experiments: Yield count

barrier(b)
yield

acquire(m)
x++
release(m)
yield

...
yield

acquire(m)
x++
release(m)
yield

acquire(m)
x=0
release(m)
yield

barrier(b)
yield

1. Static type system for verifying C-P equivalence

Cooperative correctness ∧ Coop/preemptive equivalence ⇒ Preemptive correctness

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

x is racy...

;	F	Y	M	R	L	N
F	F	Y	M	R	L	N
Y	Y	Y	Y	Y	L	L
M	M	Y	M	R	L	N
R	R	R	R	R	N	N
L	L	Y	L	—	L	—
N	N	R	N	—	N	—

t=x;   x=t+1;

<table>
<tr><td>N</td><td>N</td></tr>
<tr><td colspan="2">Error</td></tr>
</table>

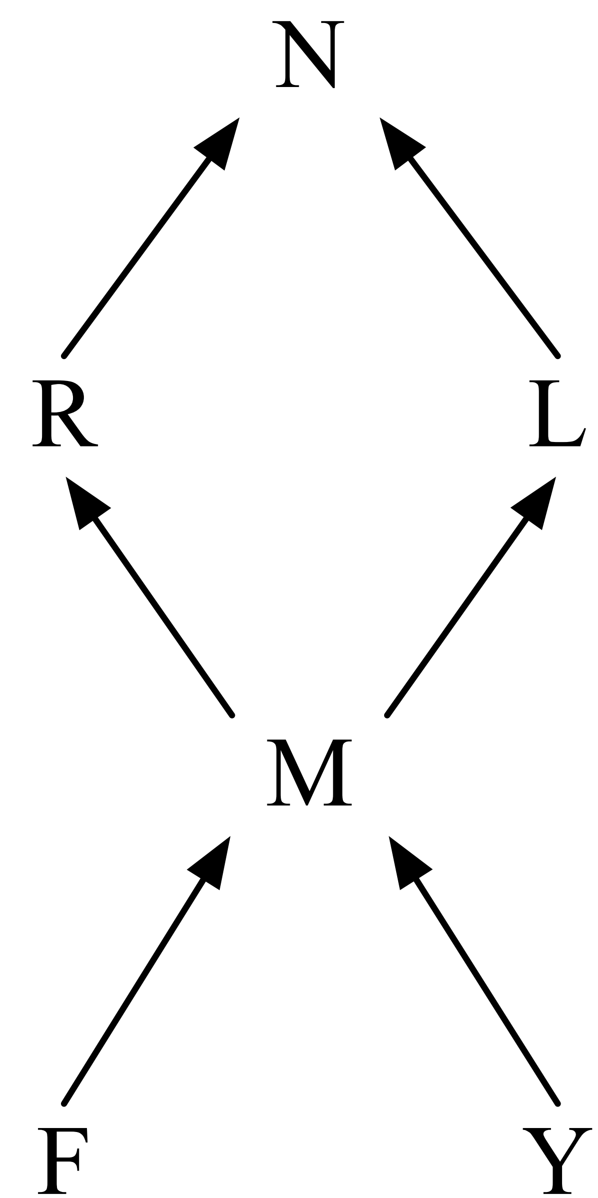
t=x; yield;   x=t+1;

<table>
<tr><td>N</td><td>Y</td><td>N</td></tr>
<tr><td colspan="2">R</td><td>N</td></tr>
<tr><td colspan="3">N</td></tr>
</table>

Effect Composition for Choice

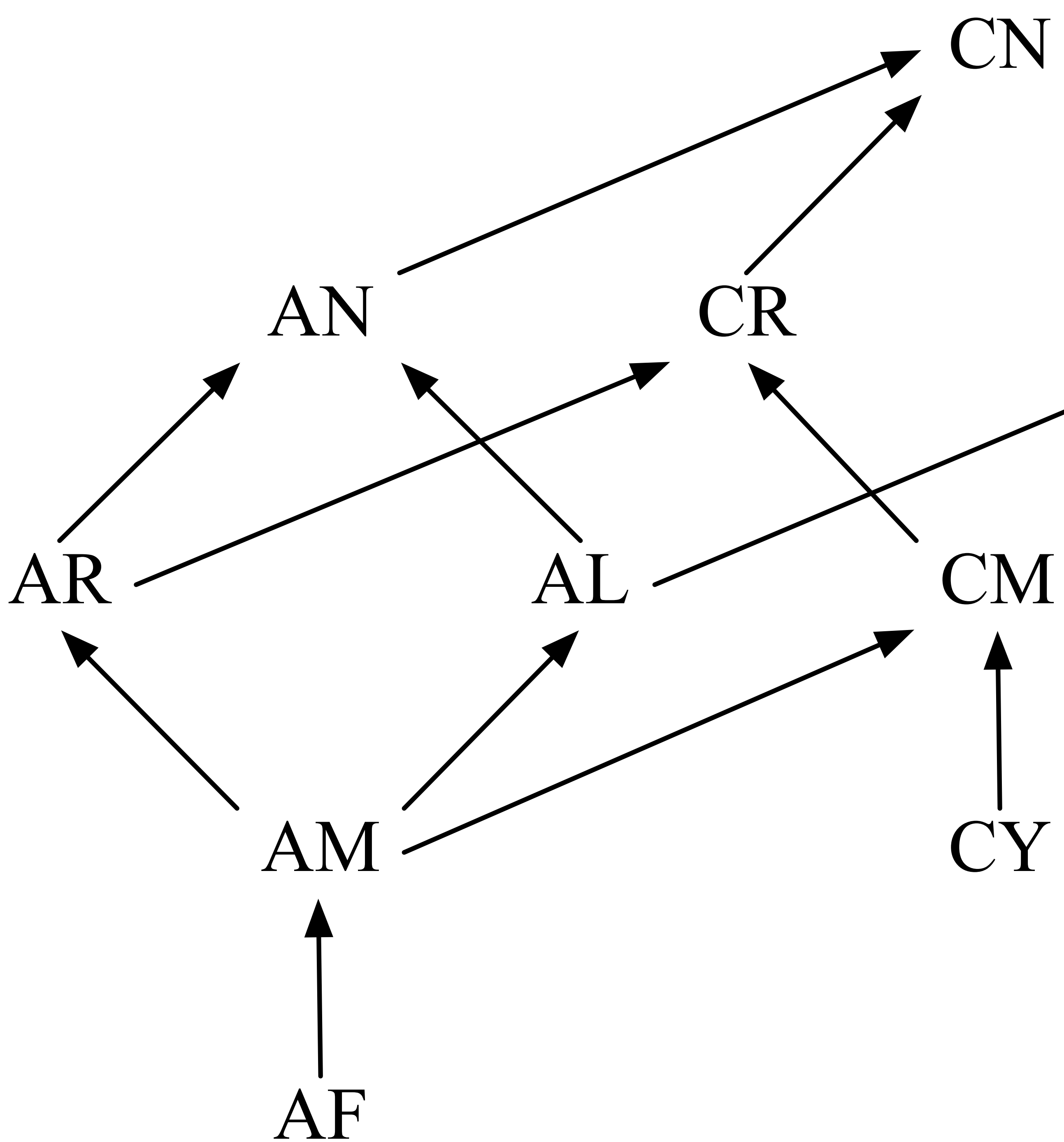
```
if (b) then acq(M); else rel(N);
```

<table>
<tr><td>R</td><td>L</td></tr>
<tr><td colspan="2">N</td></tr>
</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:
  lock ? effect : effect

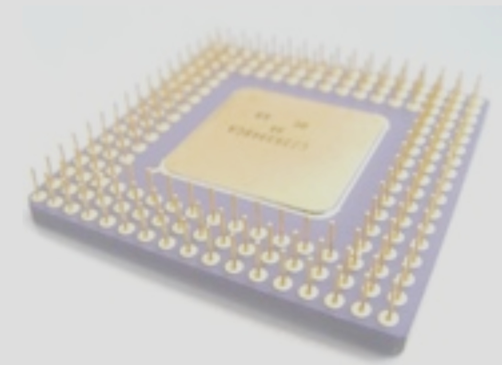
```
class StringBuffer {

    int count;

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

    ...
}
```

Cooperative Concurrency

Cooperative scheduler
seq. reasoning ok
except where yields
highlight interference

Code with sync & yields
yield
...

2. Example of coding with Yields



Preemptive scheduler
full performance
no overhead

acquire(m)
x=0
release(m)

3. Experiments: Yield count

barrier(b)
yield

acquire(m)
x++
release(m)
yield

...
yield

acquire(m)
x=0
release(m)
yield

barrier(b)
yield

acquire(m)
x++
release(m)
yield

1. Static type system for verifying C-P equivalence

Cooperativ
correctness
∧
equivalence
⇒
reemptive
correctness

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

x is volatile
concurrent calls to update_x

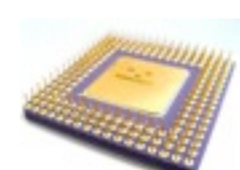
Not C-P equivalent:
No yield between accesses to x

version 1 | Coop/preemptive equivalence | $\wedge$ | Cooperative correctness | $\Rightarrow$ | Preemptive correctness

```
void update_x() {
  synchronized(m){
    x = slow_f(x);
  }
}
```

Not efficient!

high lock contention
= low performance

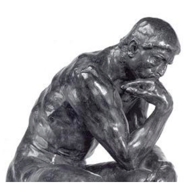

version **2** | Coop/preemptive equivalence $\wedge$ Cooperative correctness $\Rightarrow$ Preemptive correctness

```
void update_x() {
  int fx = slow_f(x);

  synchronized(m){
    x = fx;
  }
}
```

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

version **3** | Coop/preemptive equivalence | $\wedge$ | Cooperative correctness | $\Rightarrow$ | Preemptive correctness

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



Not correct:
Stale value at yield

version **4** | Coop/preemptive equivalence | $\wedge$ | Cooperative correctness | $\Rightarrow$ | 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

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

version **5** | Coop/preemptive equivalence $\wedge$ Cooperative correctness $\Rightarrow$ Preemptive correctness

```
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**

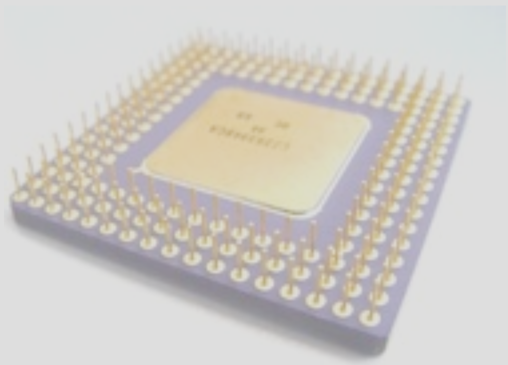
Coop/preemptive equivalence $\wedge$ Cooperative correctness $\Rightarrow$ Preemptive correctness

Cooperative Concurrency

Cooperative scheduler
seq. reasoning ok
except where yields
highlight interference

Code with sync & yields
yield
...

2. Examples of
coding with Yields



Preemptive scheduler
full performance
no overhead

acquire(m)
x=0
release(m)

3. Experiments: Yield count

barrier(b)
yield

acquire(m)
x++
release(m)
yield

acquire(m)
x=0
release(m)
yield

...
yield

barrier(b)
yield

acquire(m)
x++
release(m)
yield

1. Static type system
for verifying C-P equivalence

Cooperative correctness $\wedge$ Coop/preemptive equivalence $\Rightarrow$ Preemptive correctness

All field accesses and lock acquires

oints:

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

program	LOC	No Analysis	Method Atomic	Yields
j.u.z.Inflater	317	36	0	0
j.u.z.Deflater	381	49	0	0
j.l.StringBuffer	1276	210	9	1
j.l.String	2307	230	6	1
j.i.PrintWriter	534	73	130	26
j.u.Vector	1019	185	44	4
j.u.z.ZipFile	490	120	85	30
sparse	868	329	48	6
tsp	706	445	[illegible]	[illegible]
elevator	1447	454	[illegible]	[illegible]
raytracer-fixed	1915	565	[illegible]	[illegible]
sor-fixed	958	249	128	12
moldyn-fixed	1352	983	657	30
TOTAL	13570	3928	1890	180

Fewer interference points: less to reason about!

Summary of Cooperative Concurrency

Cooperative scheduler

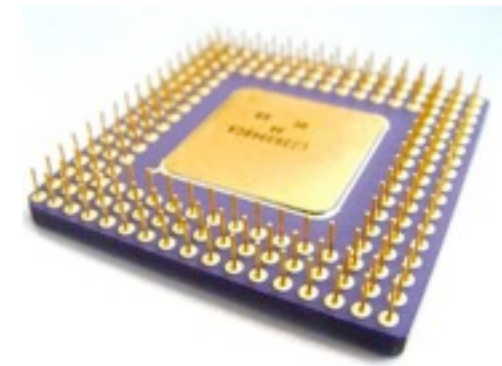
seq. reasoning ok...
...except where yields
highlight interference
x++ an increment op

	acquire(m) x=0 release(m) yield
... yield	
	barrier(b) yield
acquire(m) x++ release(m) yield	

Code with sync & yields

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

$\cong$



Preemptive scheduler

full performance
no overhead

	acquire(m) x=0 release(m) yield
... yield	barrier(b) yield
acquire(m) x++ release(m) yield	

Yields mark all thread interference

Cooperative correctness $\wedge$ Coop/preemptive equivalence $\Rightarrow$ Preemptive correctness

slang.soe.ucsc.edu/cooperability