Cooperative Concurrency for a Multicore World

Cormac Flanagan
Jaeheon Yi, Caitlin Sadowski
UCSC

Stephen Freund
Williams College

Cooperative Concurrency for a Multicore World

Cormac Flanagan
Jaeheon Yi, Caitlin Sadowski
UCSC

Stephen Freund
Williams College

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

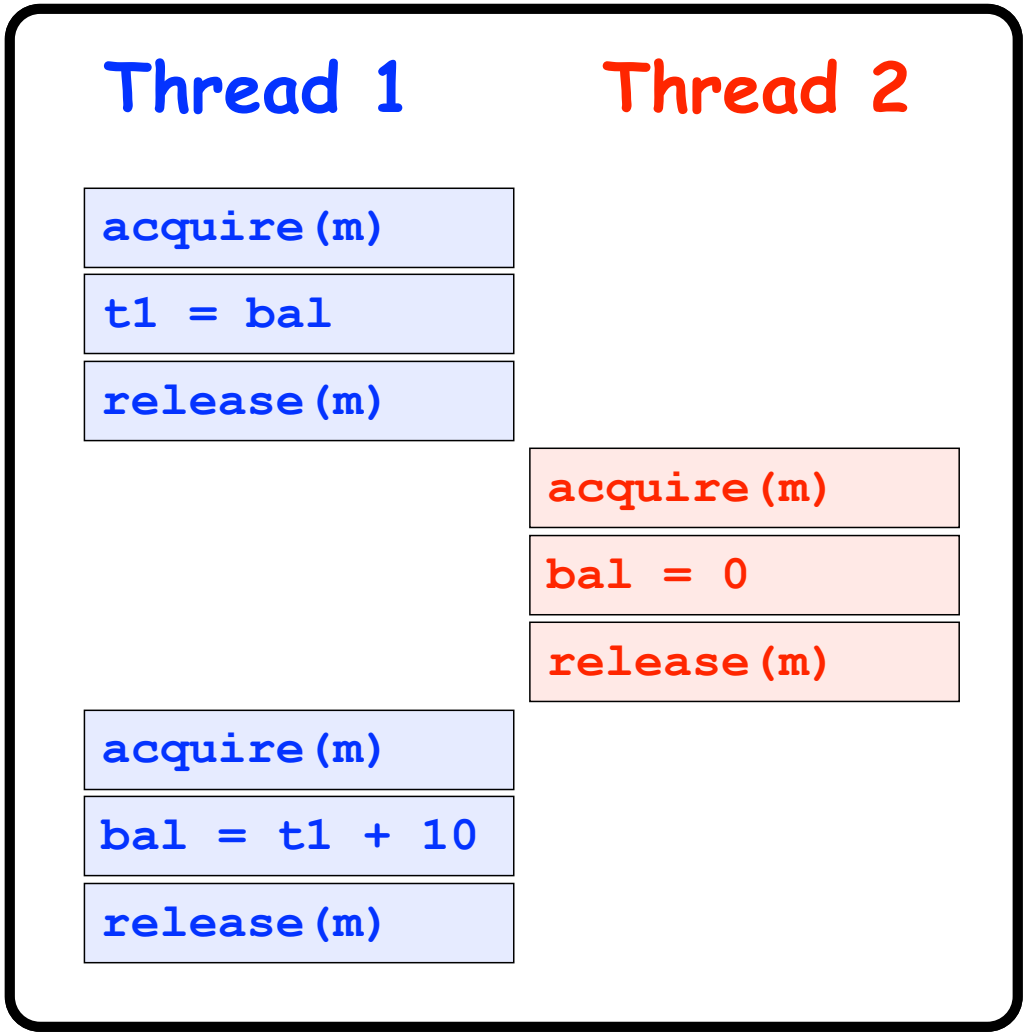
```
x = 0;
while (x < len) {
    tmp = a[x];
    b[x] = tmp;
    x++;
}
```

<u>Programmer Productivity Heuristic:</u>

assume no interference, use sequential reasoning

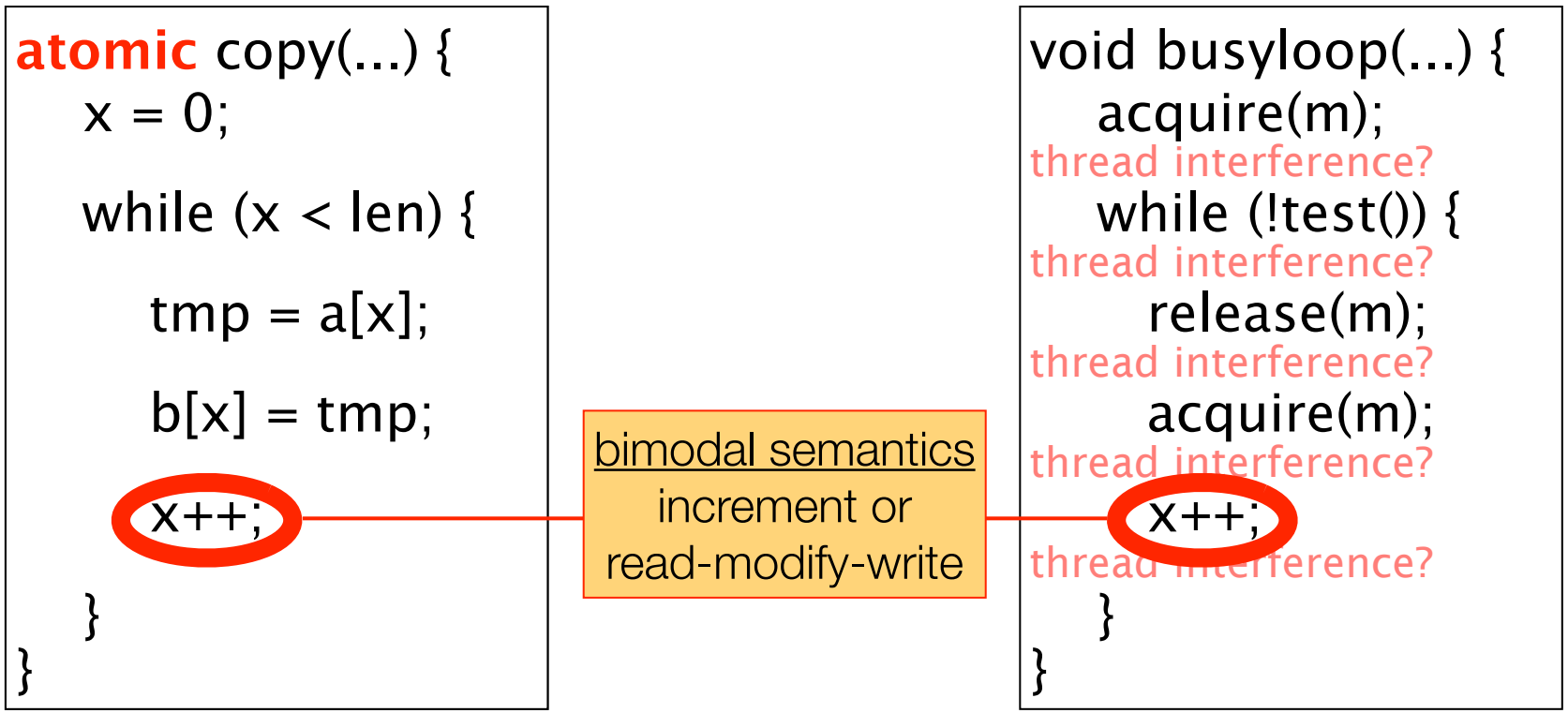
Controlling Thread Interference #2: Race Freedom

- 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

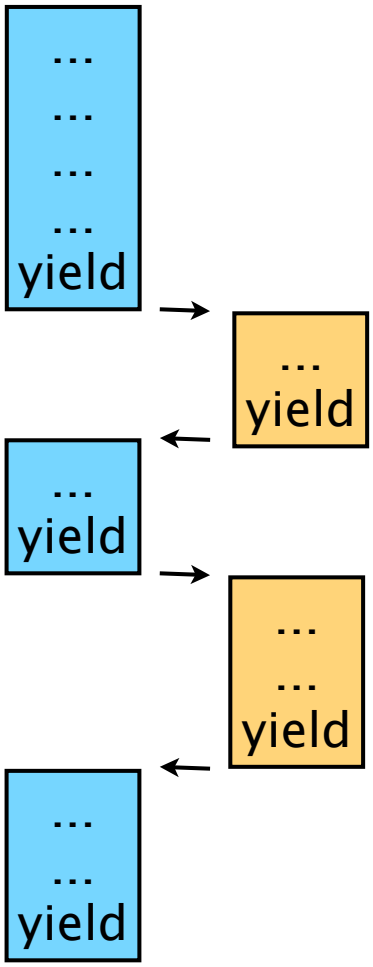
- A method is *atomic* if it behaves *as if* it executes serially, without interleaved operations of other thread



sequential reasoning ok
90% of methods atomic

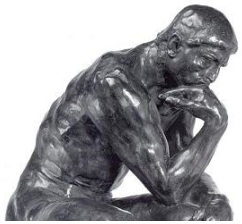
10% of methods non-atomic
local atomic blocks awkward
full complexity of threading

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

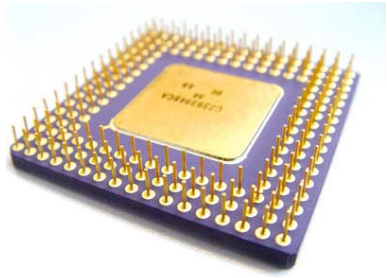
Cooperative Concurrency



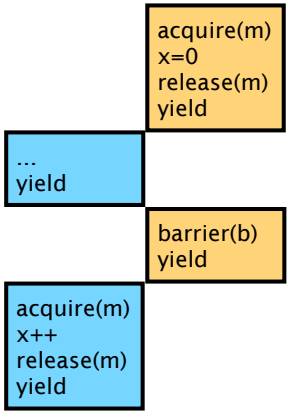
Cooperative scheduler
seq. reasoning ok
except where yields
highlight interference

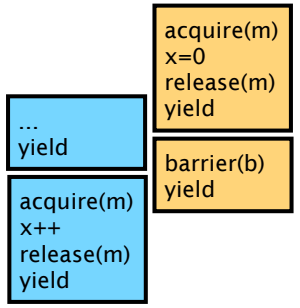
Code with sync & yields

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



Preemptive scheduler
full performance
no overhead



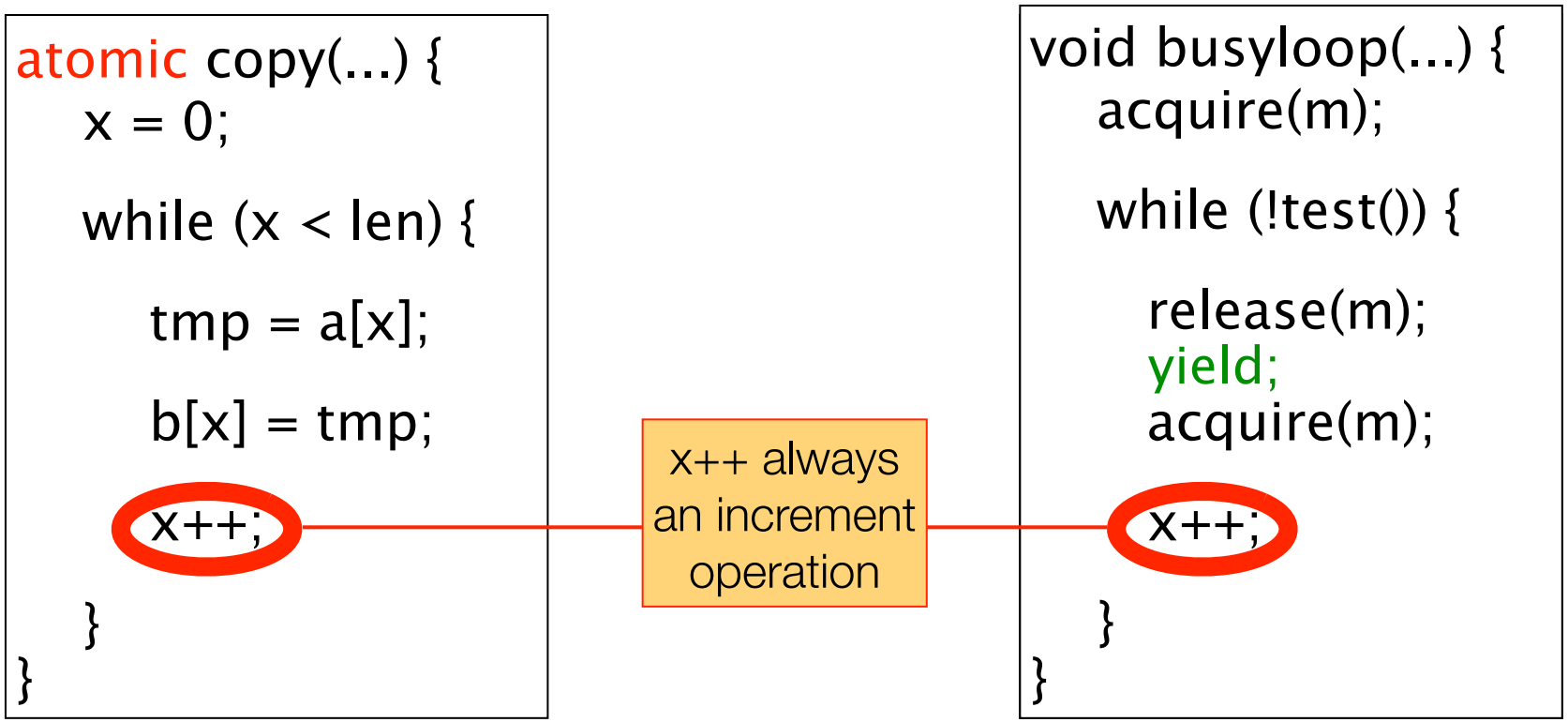


Yields mark all thread interference

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

Benefits of Yield over Atomic

- Atomic methods are exactly those with no yields



atomic 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++

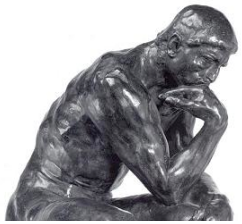
```
x = 0;

while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

}
```

Cooperability in the design space

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

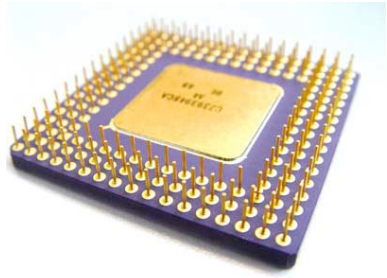
Transactional Memory, Larus & Rajwar, 2007
Automatic mutual exclusion, Isard & Birrell, HOTOS '07

Cooperative Concurrency

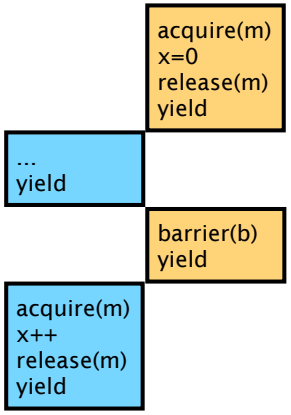


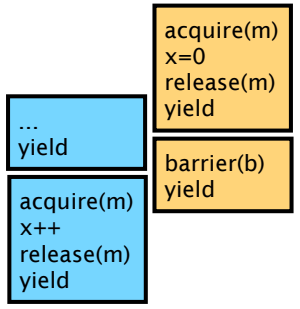
Cooperative scheduler
seq. reasoning ok
except where yields
highlight interference

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



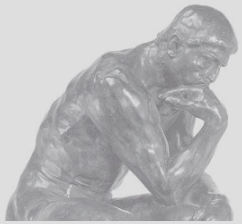
Preemptive scheduler
full performance
no overhead





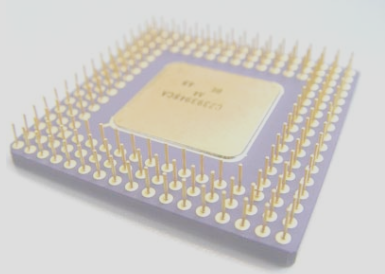
Cooperative correctness $\wedge$ Coop/preemptive equivalence $\Rightarrow$ Preemptive correctness

Cooperative Concurrency



Cooperative scheduler
seq. reasoning ok
except where yields

Code with sync & yields
...
yield
...



Preemptive scheduler
full performance
no overhead

1. Examples of coding with Yields

2. User study: Do Yields help?

...
yield
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)

3. Dynamic analysis for C-P equivalence (detecting missing yields)

Cooperati correctne

4. Static type system for verifying C-P equivalence

Preemptive correctness

Example: java.util.StringBuffer.append(...)

```
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() {...}
```

```
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

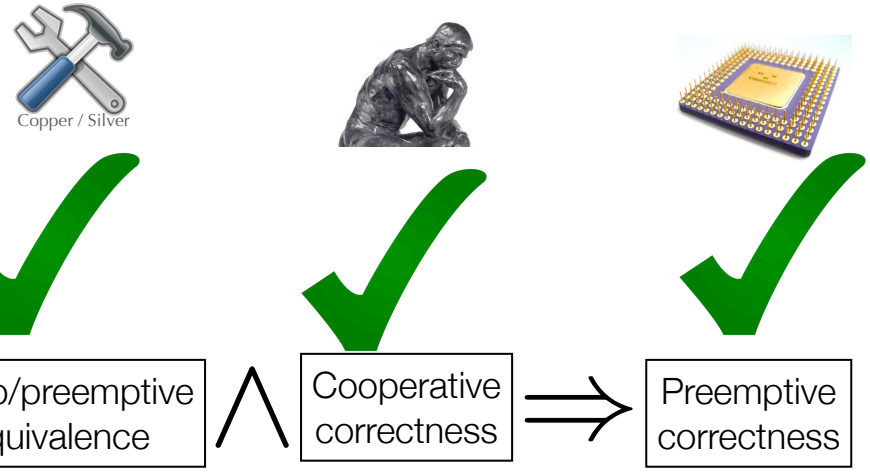
version **1**

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

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

Not efficient!

high lock contention
= low performance



version **2**

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

   acquire(m);
   x = fx;
   release(m);
}
```

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;
   acquire(m);
   x = fx;
   release(m);
}
```

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;
    }
  }
}
```

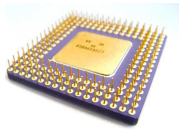
restructure:
test and retry pattern

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

version **5**

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

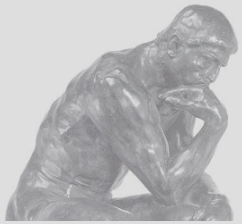
```
void update_x() {
  int y = x;
  for (;;) {
    yield;
    int fy = slow_f(y);
    acquire(m);
    if (x == y) {
      x = fy;
      return;
    } else {
      y = x;
    }
    release(m);
  }
}
```



version **6**

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

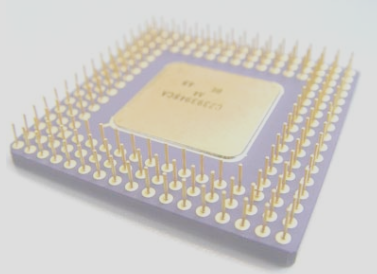
Cooperative Concurrency



Cooperative scheduler
seq. reasoning ok
except where yields

Code with sync & yields
...
yield
...

1. Examples of coding with Yields



Preemptive scheduler
full performance
no overhead

2. User study:
Do Yields help?

...
yield

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)

3. Dynamic analysis for C-P equivalence
(detecting missing yields)

Cooperati
correctne

4. Static type system
for verifying C-P equivalence

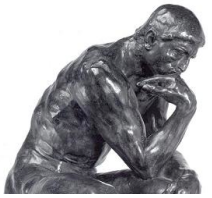
Preemptive
correctness

A Preliminary User Study of Cooperability

- Hypothesis: Yields help code comprehension + defect detection?

- Study structure
  - Web-based survey, background check on threads
  - Between-group design - code with or without yields
  - Three code samples, based on real-world bugs
  - Task: Identify all bugs

User Evaluation for Cooperability



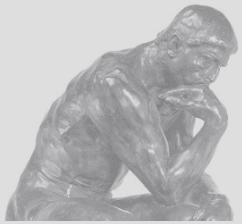
StringBuffer	Concurrency bug	Some other bug	Didn't find bug	Total
Yields	10	1	1	12
No Yields	1	5	9	15

All Samples	Concurrency bug	Some other bug	Didn't find bug	Total
Yields	30	3	3	36
No Yields	17	6	21	44

- Difference is statistically significant ($p < 0.001$)

User Evaluation of Correctness Conditions: A Case Study of Cooperability. Sadowski & Yi, PLATEAU 2010

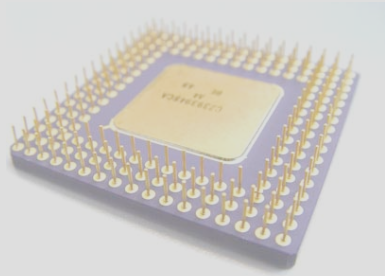
Cooperative Concurrency



Cooperative scheduler
seq. reasoning ok
except where yields

Code with sync & yields
...
yield
...

1. Examples of coding with Yields



Preemptive scheduler
full performance
no overhead

2. User study:
Do Yields help?

...
yield

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)

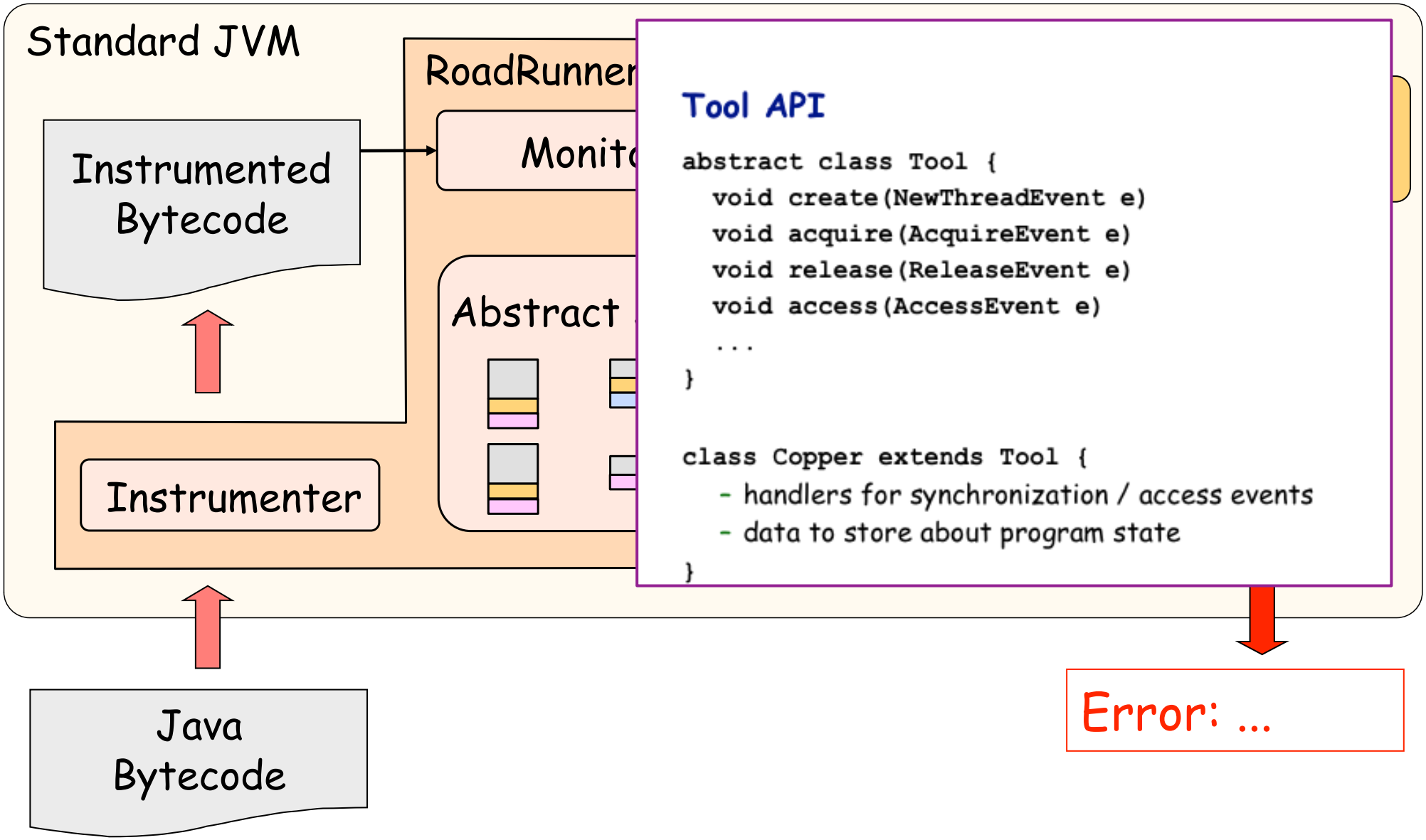
3. Dynamic analysis for C-P equivalence
(detecting missing yields)

Cooperati
correctne

4. Static type system
for verifying C-P equivalence

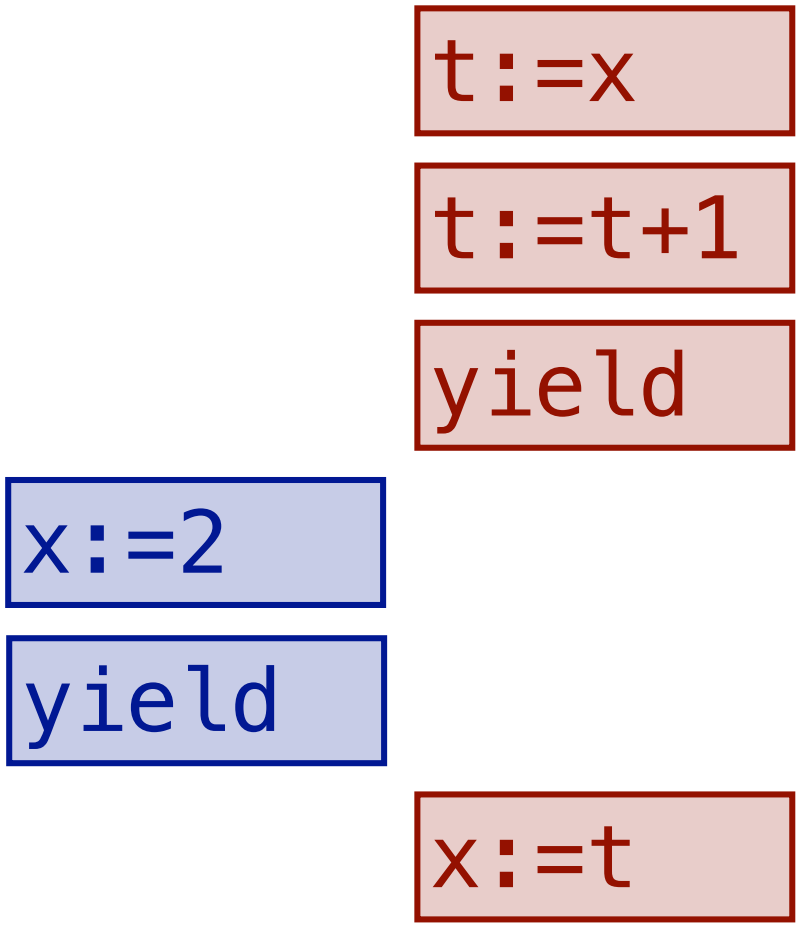
Preemptive
correctness

RoadRunner Framework for Dyanamic Concurrency Analyses [PASTE '10, github]



Others: Sofya [KDR 07], CalFuzzer JNPS 09]

cooperative trace:
context switch at yields



cooperative trace:
context switch at yields

```
t:=x
t:=t+1
yield
```

```
x:=2
yield
```

```
x:=t
```

preemptive trace:
context switch anywhere

```
t:=x
```

```
x:=2
```

```
t:=t+1
yield
```

```
yield
```

```
x:=t
```

program is **C-P equivalent** if any preemptive trace is equivalent to some cooperative trace

Concurrency Control and Recover in Database Systems. Bernstein, Hadzilacos, Goodman, 1987

cooperative trace:
context switch at yields

```
t:=x
t:=t+1
yield
```

```
x:=2
yield
```

```
x:=t
```

preemptive trace:
context switch anywhere

```
t:=x
```

```
x:=2
```

```
t:=t+1
yield
```

```
yield
```

```
x:=t
```

program is **C-P equivalent** if any preemptive trace is equivalent to some cooperative trace

Concurrency Control and Recover in Database Systems. Bernstein, Hadzilacos, Goodman, 1987

COPPER detects coop/preemptive violations

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

Transaction is code between two yields

```
acq m
rd x 2
rel m



acq m
rd x 1
rel m
...
```

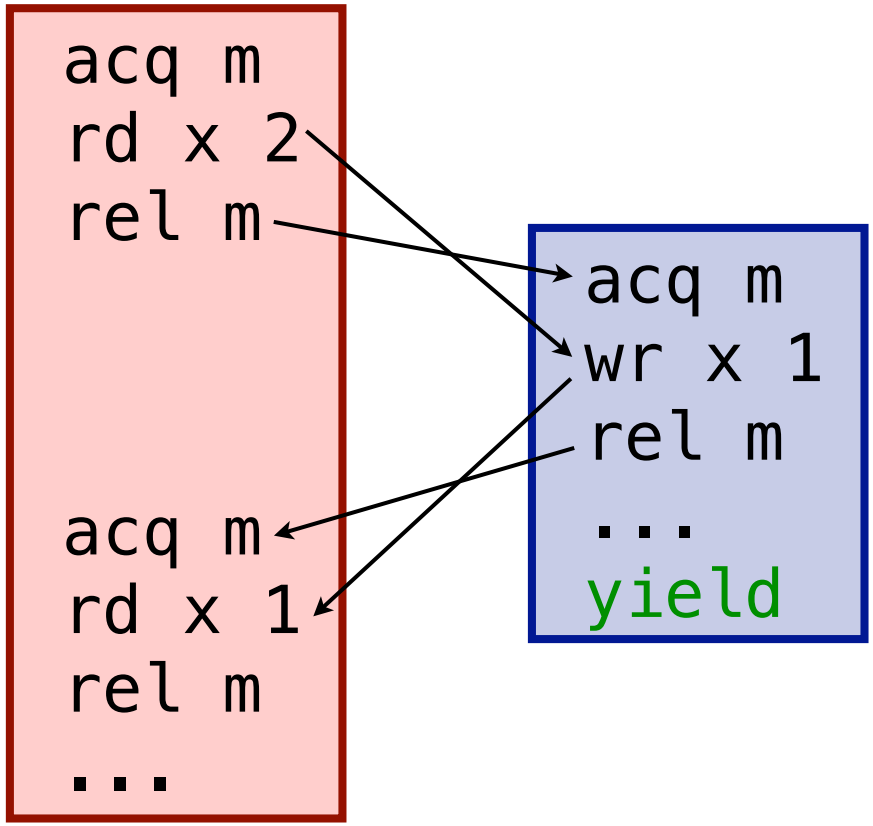
```
acq m
wr x 1
rel m
...
yield
```

Cooperative Reasoning for Preemptive Execution. Yi Sadowski, Flanagan, PPOPP'11.

COPPER detects cooperability violations

Happens-before order

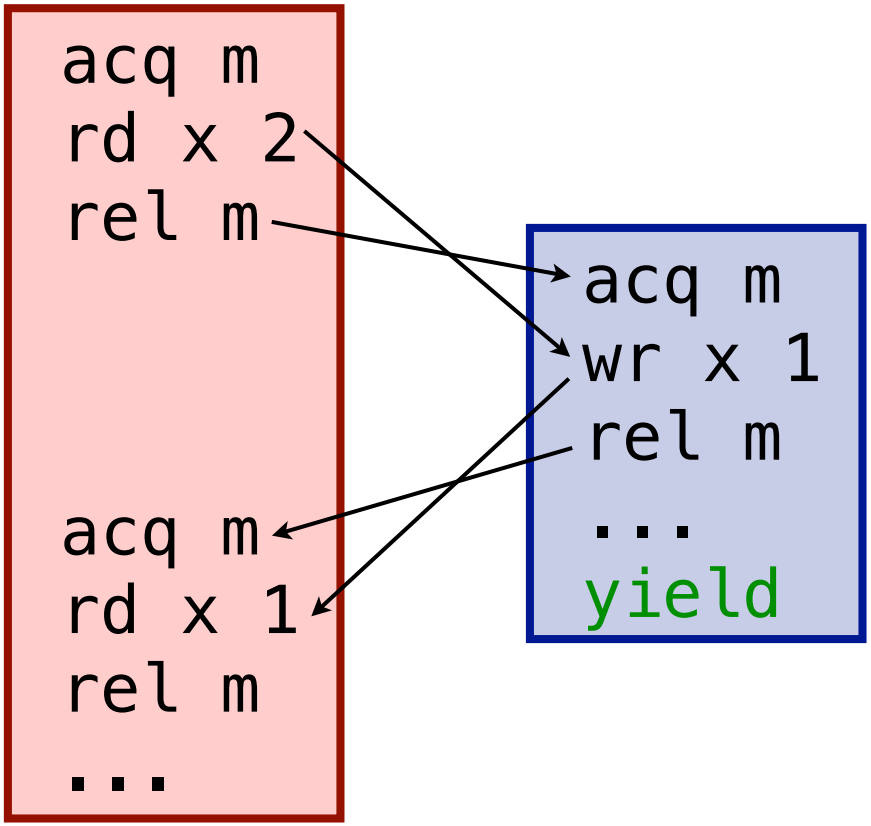
- program order
- synchronization order
- communication order



Cooperative Reasoning for Preemptive Execution. Yi Sadowski, Flanagan, PPOPP'11.

COPPER detects cooperability violations


```
yield;
acquire(m);
while(x>0){
  release(m); //missing yield!
  acquire(m);
}
assert x==0;
release(m);
yield;
```



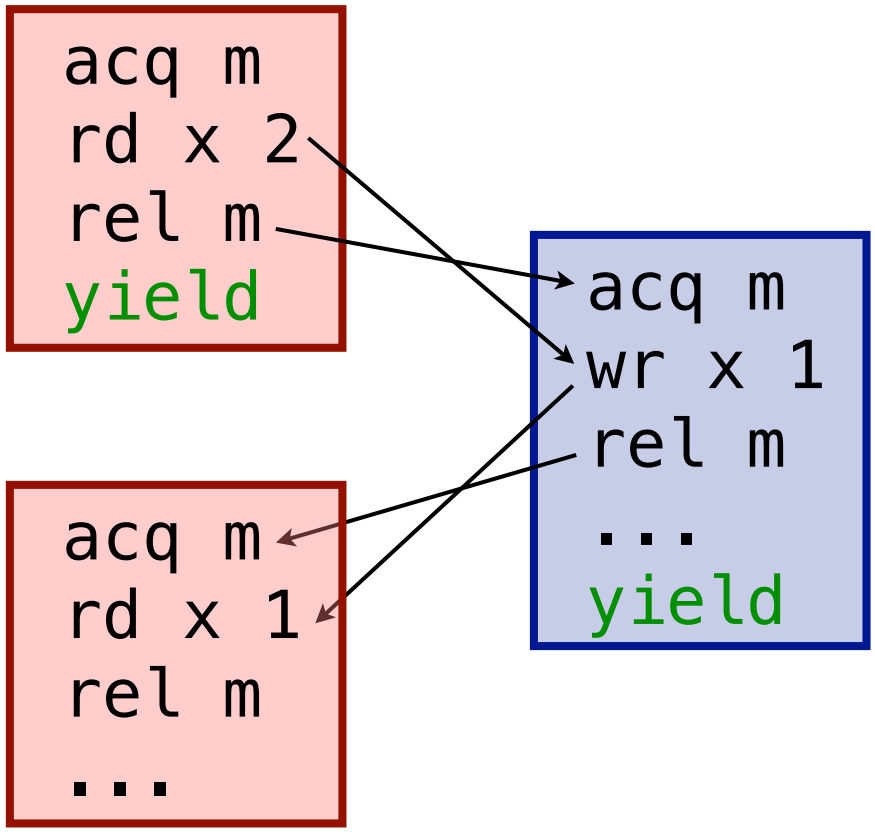
Error: Cycle implies missing yield

Cooperative Reasoning for Preemptive Execution. Yi Sadowski, Flanagan, PPOPP'11.

COPPER detects cooperability violations

Transactional HB order has no cycles if and only if trace is cooperative-preemptive equivalent

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



Cooperative Reasoning for Preemptive Execution. Yi Sadowski, Flanagan, PPOPP'11.

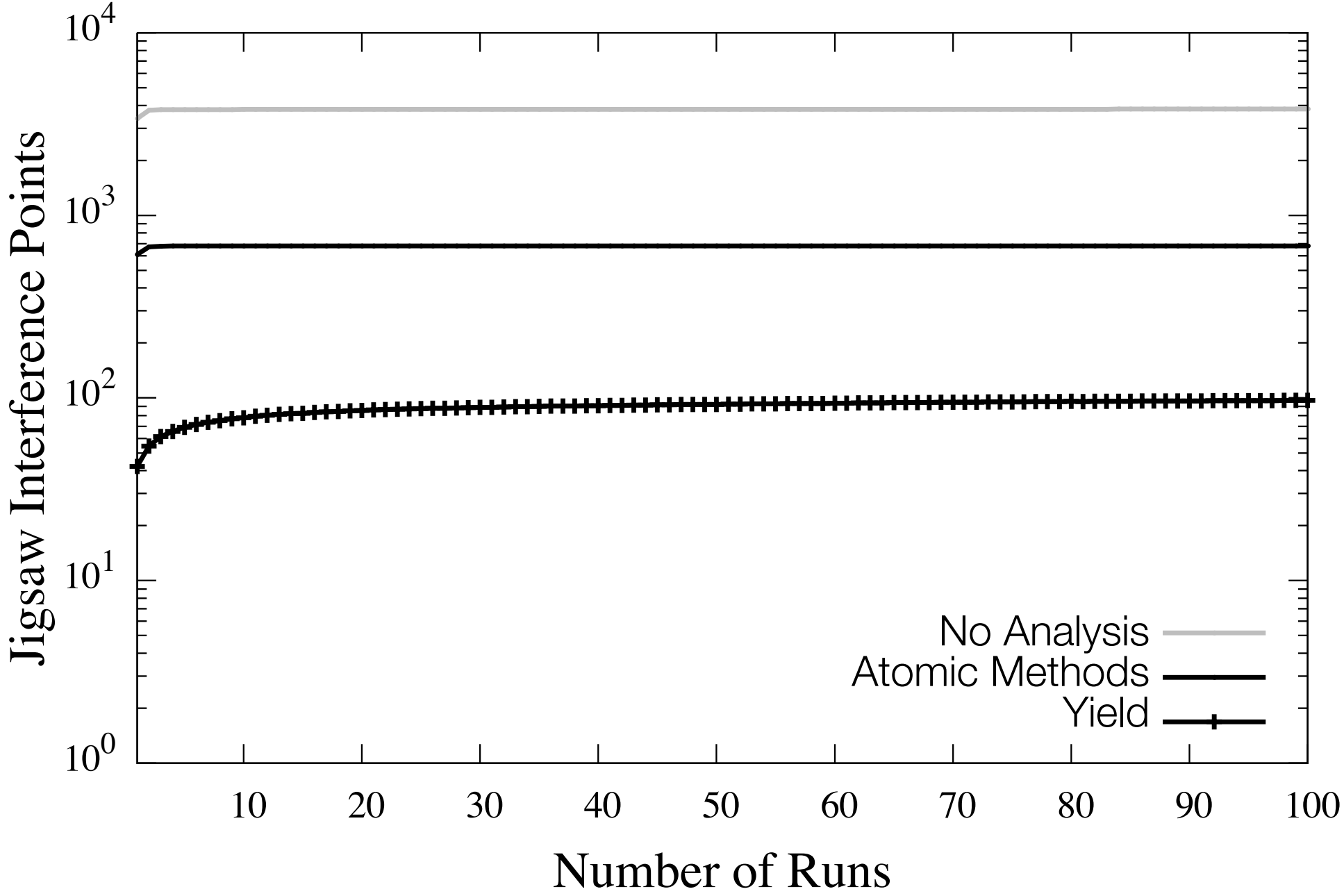
All field accesses and lock acquires

Results

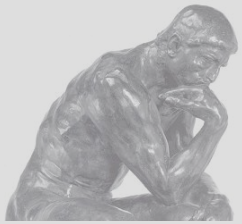
In non-atomic methods, count field accesses, lock acquires, and atomic methods calls

program	LLOC	No Analysis	Atomic Methods	Yields
sparse	712	196	49	0
sor	721	134	49	3
series	811	90	31	0
crypt	1083	252	55	0
moldyn	1299	737	64	3
elevator	1447	247	54	3
lufact	1472	242	57	3
raytracer	1862	355	65	3
montecarlo	3557	377		
hedc	6409	305		
mtrt	6460	695		
raja	6863	396	45	0
colt	25644	601	113	13
jigsaw	48674	3415	550	47

Fewer interference points: less to reason about!


Jigsaw Interference Points
10^4
10^3
10^2
10^1
10^0
10
20
30
40
50
60
70
80
90
100
Number of Runs
No Analysis
Atomic Methods
Yield

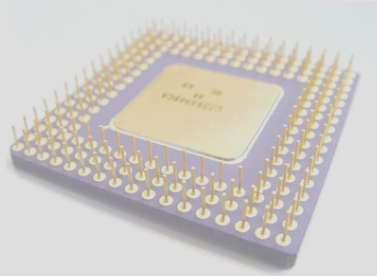
Cooperative Concurrency



Cooperative scheduler
seq. reasoning ok
except where yields

Code with sync & yields

yield
...



Preemptive scheduler
full performance
no overhead

1. Examples of coding with Yields

2. User study: Do Yields help?

...
yield

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)

3. Dynamic analysis for C-P equivalence (detecting missing yields)

Cooperati correctne

4. Static type system for verifying C-P 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)

- Well-typed programs are cooperative-preemptive equivalent

Effect Language

- Approach: Compute an *effect* for each program expression/statement that summarizes how that computation interact with other threads
- Effects:
  - R right-mover lock acquire
  - L left-mover lock release
  - B 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)

Example: TSP algorithm

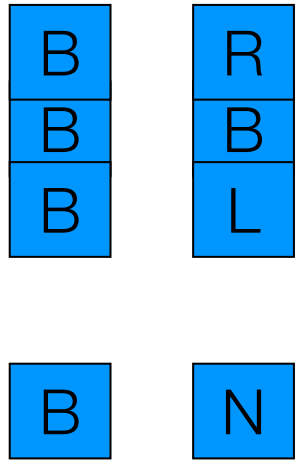
```
Object lock;
volatile int shortestPathLength; // lock held for writes

both-mover void searchFrom(Path path) {
  yield
  if (path.length >= shortestPathLeng return;

  if (path.isComplete()
    yield;
    synchronized( ck) {
      if (path.length < shortestPathLeng
        shortestPathLeng = path.leng
    }
  } else {
    for (Path c : path.children()
      searchFrom(c)
  }
}
```

Y B N B Y R B B N B L B B

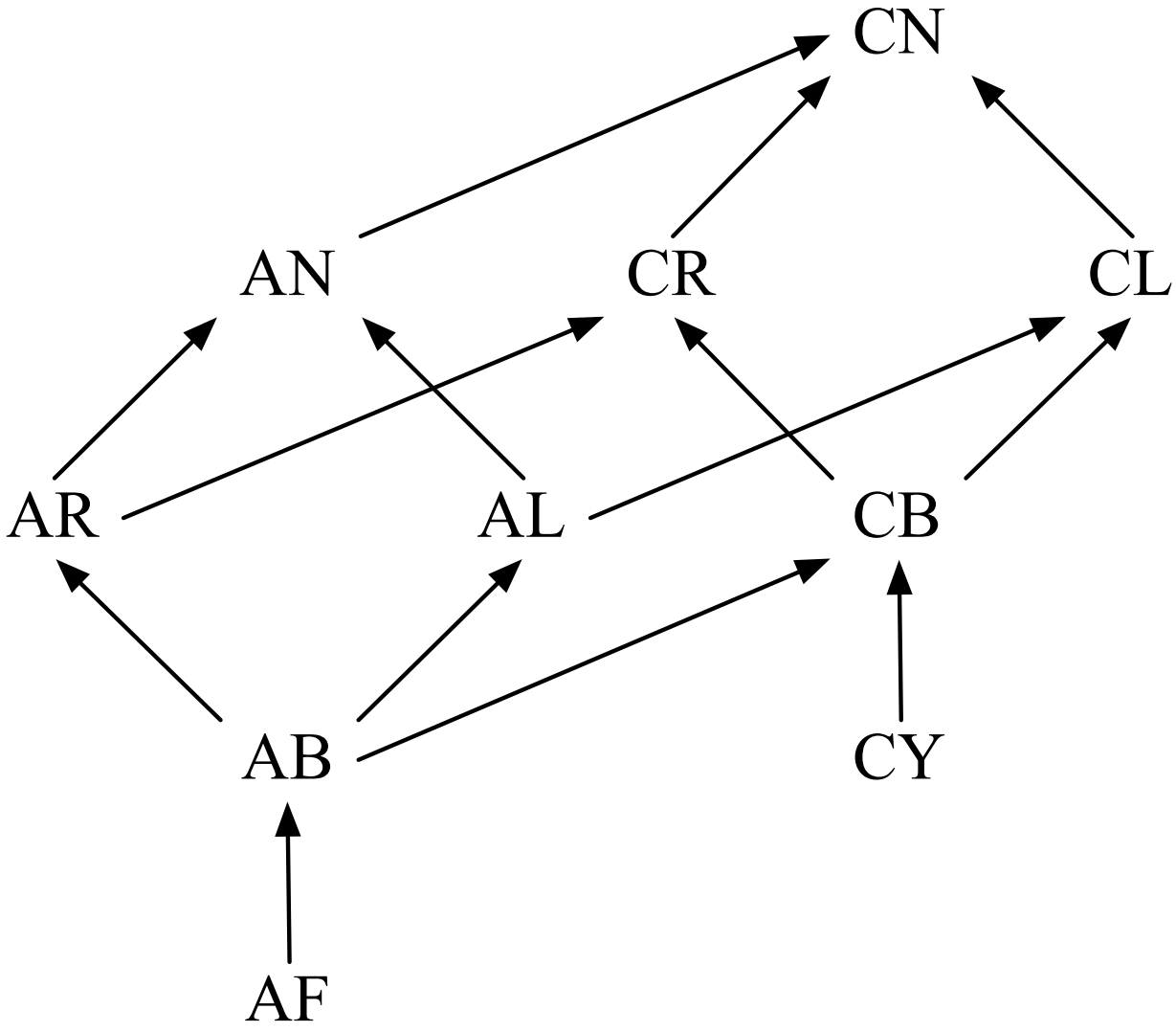
Match pattern (R* [N] L* Y)* (R* [N] L*)

Conditional Effects

```
class StringBuffer {

    int count;

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

    ...
}
```



Full Effect Lattice



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	38	0	0
j.u.z.Deflater	381	44	0	0
j.l.StringBuffer	1276	207	9	1
j.l.String	2307	154	5	1
j.i.PrintWriter	534	54	69	26
j.u.Vector	1019	183	19	1
j.u.z.ZipFile	490	81	69	30
sparse	868	231	41	8
tsp	706			
elevator	1447			
raytracer-fixed	1915			
sor-fixed	958	200	137	13
moldyn-fixed	1352	922	651	25
TOTAL	13570	3284	1595	175

Fewer interference points: less to reason about!

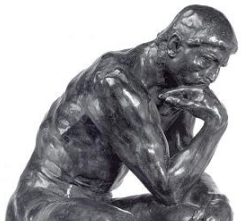
A More Precise Yield Annotation

```
Object lock;
volatile int shortestPathLength;

compound both-mover 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())
      searchFrom(c);
  }
}
```

A More Precise Yield Annotation

```
Object lock;
volatile int shortestPathLength;

compound both-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.children())
      searchFrom#(c);
  }
```

Summary of Cooperative Concurrency



Cooperative scheduler

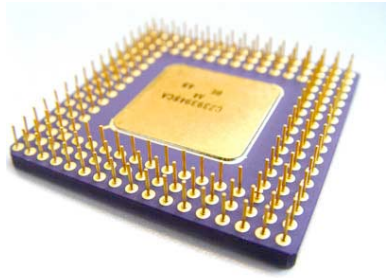
seq. reasoning ok...

...except where yields highlight interference

x++ an increment op

Code with sync & yields

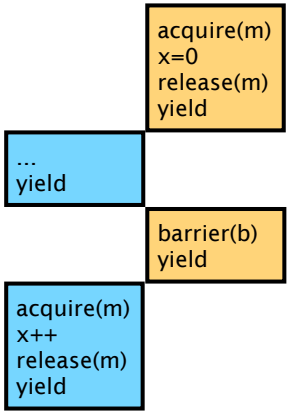
```
...
acquire(m)
x++
release(m)
yield   // interference
...
```

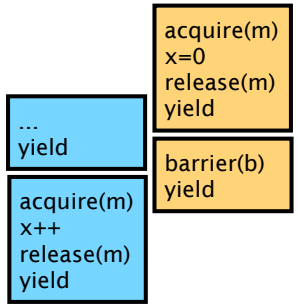


Preemptive scheduler

full performance

no overhead





Yields mark all thread interference

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

Summary

- Thread interference notoriously problematic in multithreaded code
  - Ugly semantics, awkward to reason about correctness
  - Destructive interference syntactically hidden, often ignored

- Proposed approach
  - Document interference with yields (few required, 1-10/KLOC)
  - Analysis tools verify cooperative-preemptive equivalence
    - Preemptive scheduling for execution: full performance
    - Cooperative scheduling for reasoning about correctness
      - Sequential reasoning by default
      - Yields highlight thread interference, helps detect concurrency bugs

slang.soe.ucsc.edu/cooperability

```
void update_x() {

  boolean done = false;
  int y = x;




  while ( !done ) {
    yield;
    int fy = f(y);
    acquire(m);
    if (x == y) {
      x = fy;
      done = true;
    } else {
      y = x;
    }
    release(m);

  }
}
```

(a) Using yield annotations

```
void update_x() {

  boolean done = false;
  int y = x;




  while ( !done ) {
    atomic {
      int fy = f(y);
      acquire(m);
      if (x == y) {
        x = fy;
        done = true;
      } else {
        y = x;
      }
      release(m);
    }
  }
}
```

(b) Using one atomic block annotation

```
void update_x() {

  boolean done;
  int y;
  atomic {
    done = false;
    y = x;
  }

  while ( atomic { !done } ) {
    atomic {
      int fy = f(y);
      acquire(m);
      if (x == y) {
        x = fy;
        done = true;
      } else {
        y = x;
      }
      release(m);
    }
  }
}
```

(c) Using three atomic block annotations