Analysis Techniques to Detect Concurrency Errors (part 2)

Cormac Flanagan
UC Santa Cruz

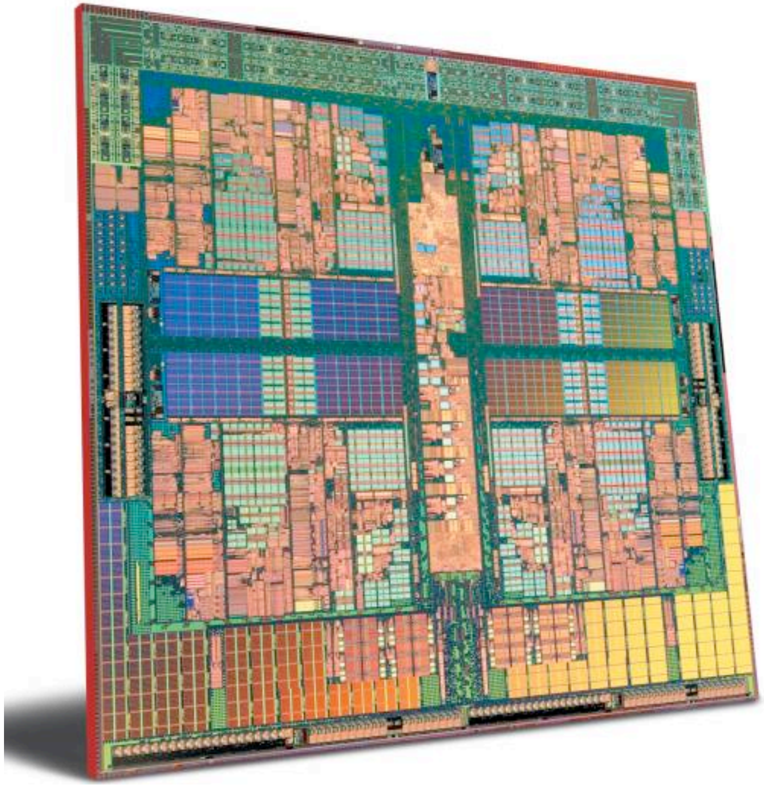
Stephen Freund
Williams College

Student Contributors

- 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

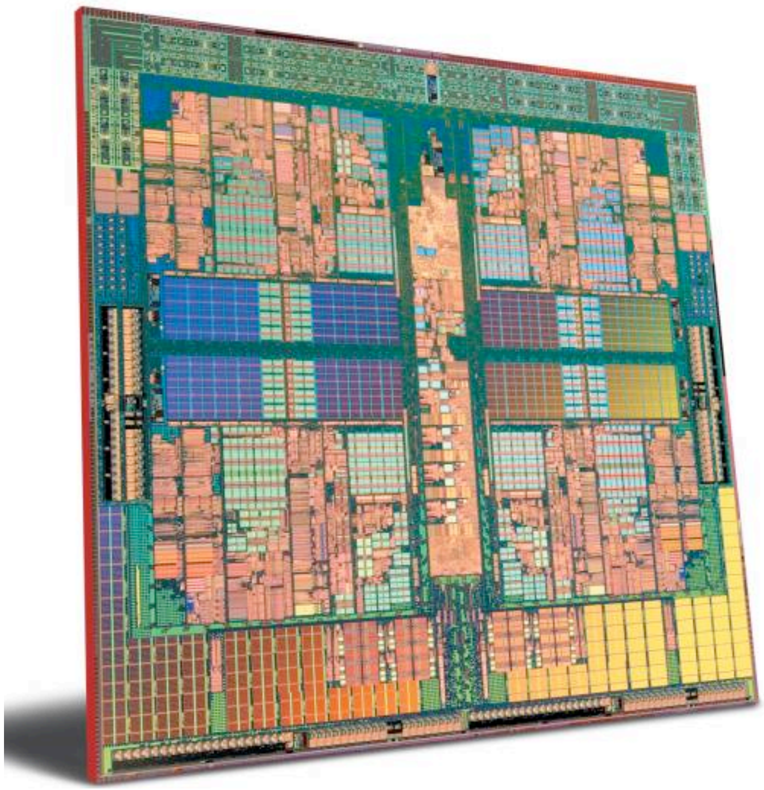
- Ben Wood, Williams College (now at Wellesley College)
- Diogenes Nunez, Williams College (now at Tufts)
- Antal Spector-Zabusky, Williams College (now at UPenn)
- James Wilcox, Williams College (now at UW)
- Parker Finch, Williams College
- Emma Harrington, Williams College

Multithreading and Multicore CPUs



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

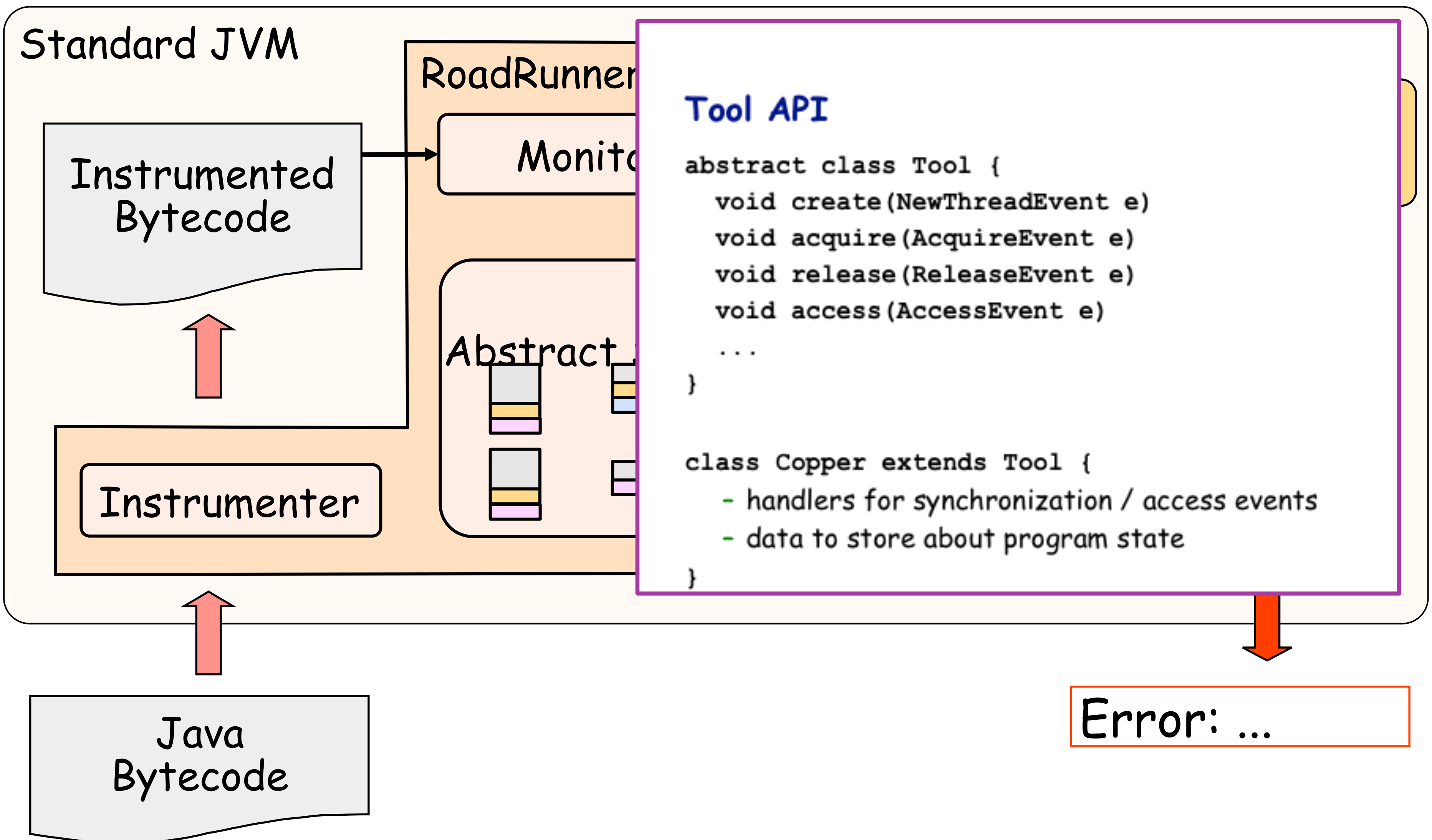
Controlling Thread Interference



Static Techniques:
- Types for Race Detection
- *Types for Atomicity*

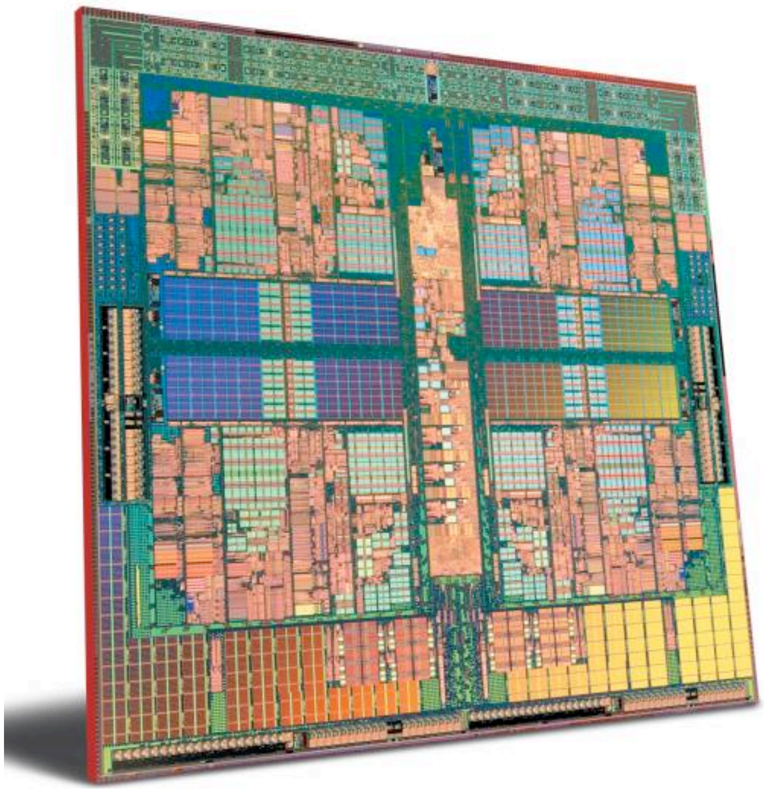
Dynamic Techniques:
- Data Races
- *Atomicity / Serializability*
- *Cooperative Concurrency*

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



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

Controlling Thread Interference



Static Techniques:

- Types for Race Detection
- *Types for Atomicity*

Dynamic Techniques:

- Data Races
- *Atomicity / Serializability*
- *Cooperative Concurrency*

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?
 }
```

1 Inspect code

2 Identify where interference does not occur

```
 x = 0;

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

    x++;

 }
```

Controlling Thread Interference: #1 Manually w/ "Productivity Heuristic"

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

1 Assume no interference

2 Use sequential reasoning

```
x = 0;

while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

}
```

- Works some of the time, but subtle bugs...

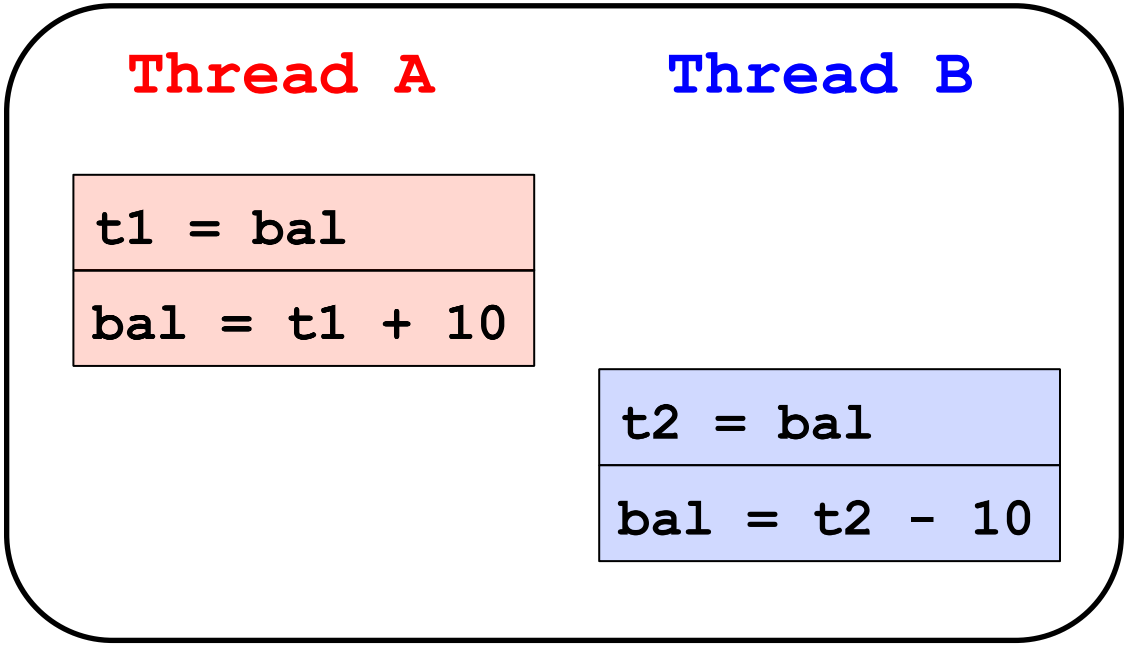
Controlling Thread Interference: #2 Enforce Race Freedom

- Race Conditions

  two concurrent unsynchronized accesses, at least one write

```
Thread A
  ...
  t1 = bal;
  bal = t1 + 10;
  ...
```

```
Thread B
  ...
  t2 = bal;
  bal = t2 - 10;
  ...
```



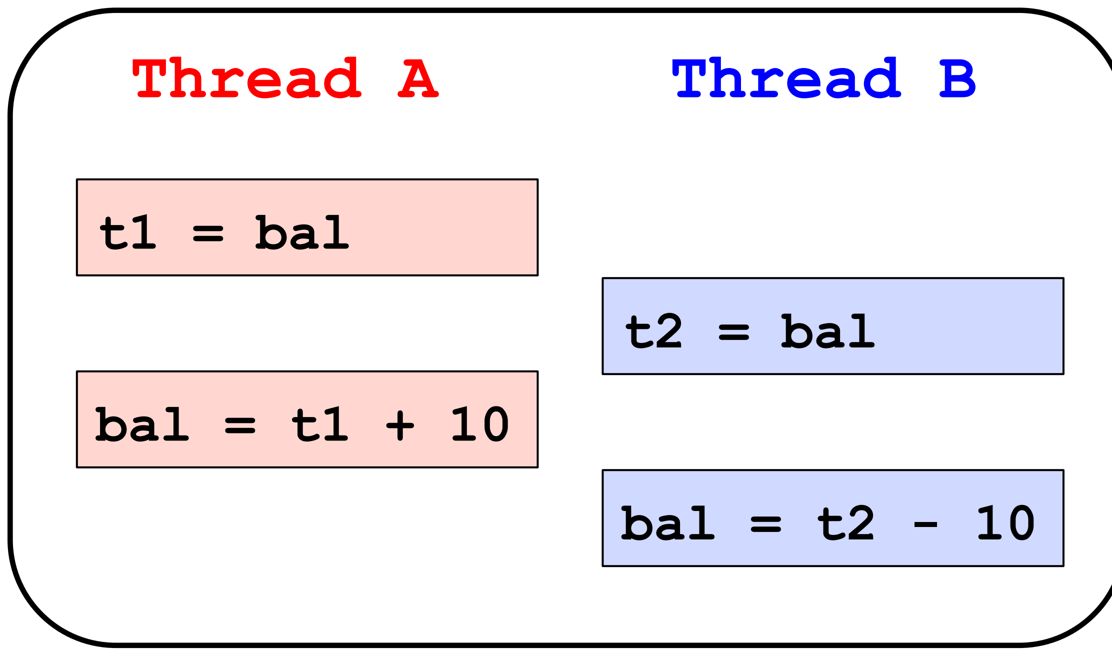
Controlling Thread Interference: #2 Enforce Race Freedom

- Race Conditions

  two concurrent unsynchronized accesses, at least one write

```
Thread A
  ...
  t1 = bal;
  bal = t1 + 10;
  ...
```

```
Thread B
  ...
  t2 = bal;
  bal = t2 - 10;
  ...
```



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

- 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 = 0
  rel(m);
```

Thread A

```
acq(m)
t1 = bal
rel(m)
```

Thread B

```
acq(m)
bal = 0
rel(m)
```

Thread A

```
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

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

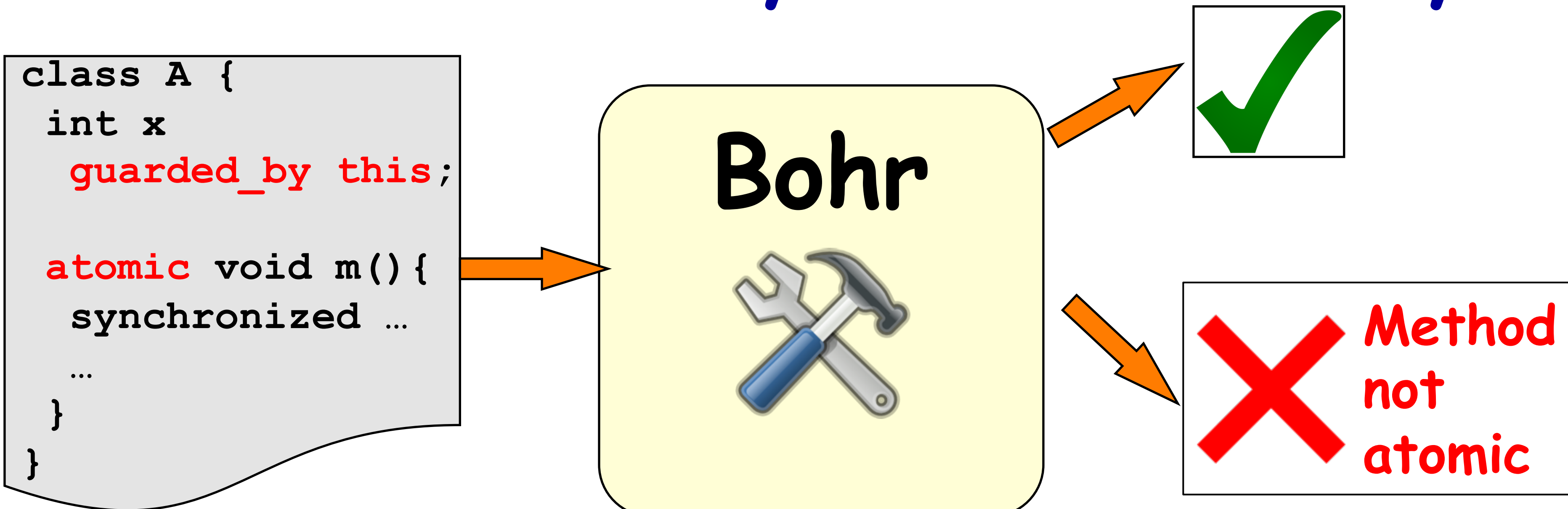
Controlling Thread Interference: #3 Enforce Atomicity

Atomic method must behave as if it executed serially, without interleaved operations of other thread

```
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

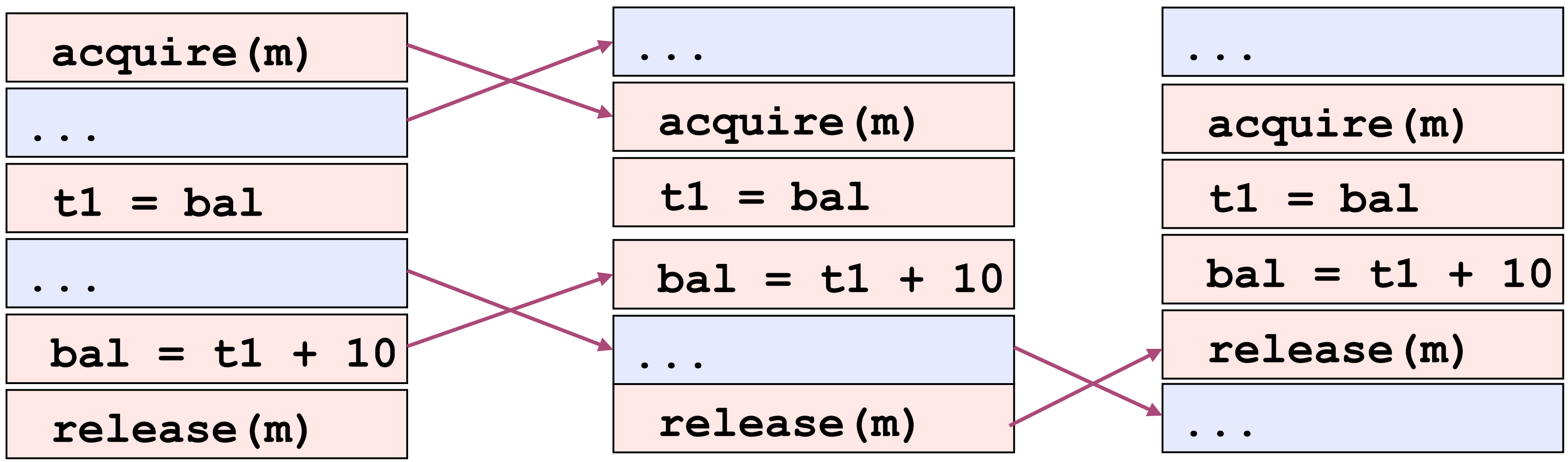
Bohr: Static Analysis for Atomicity



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



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*

Examples

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

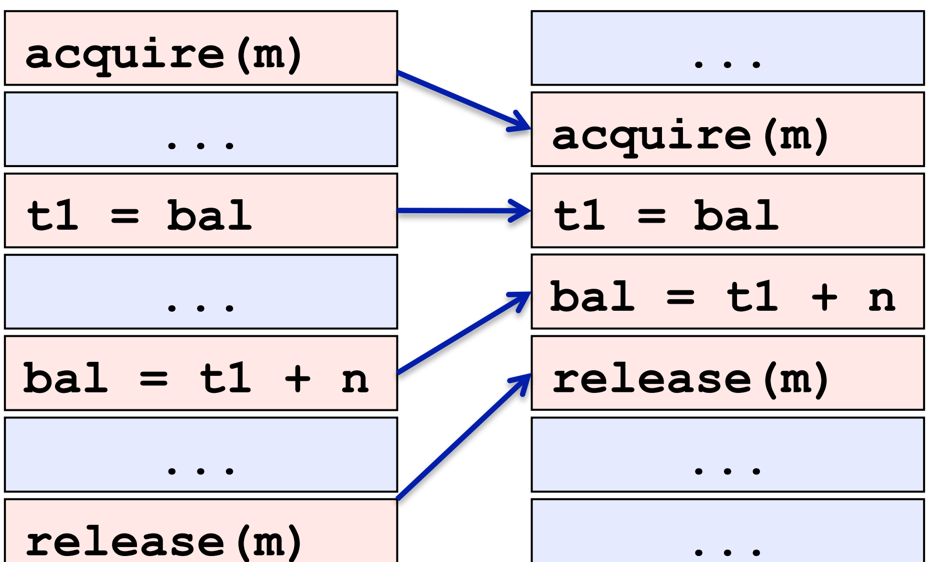
R

M

M

L

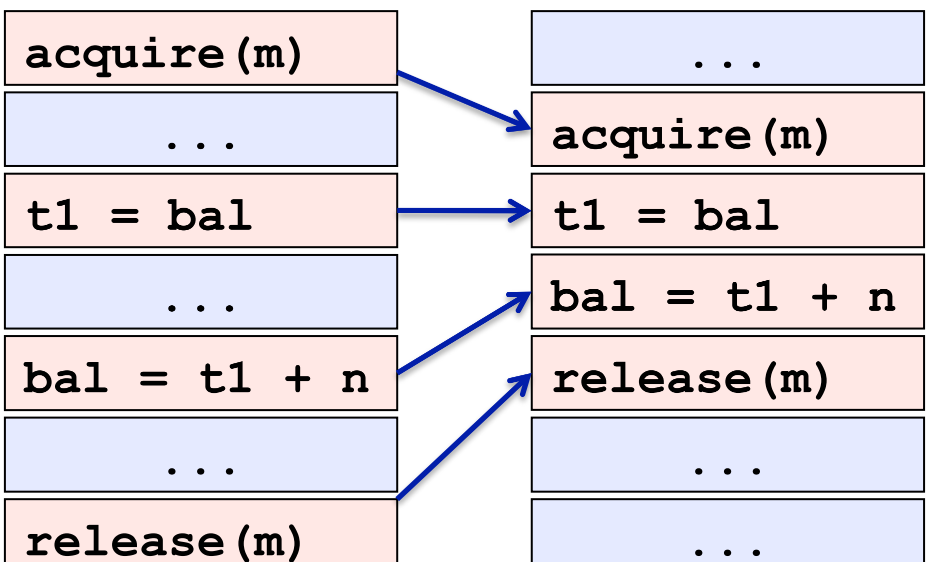
✓



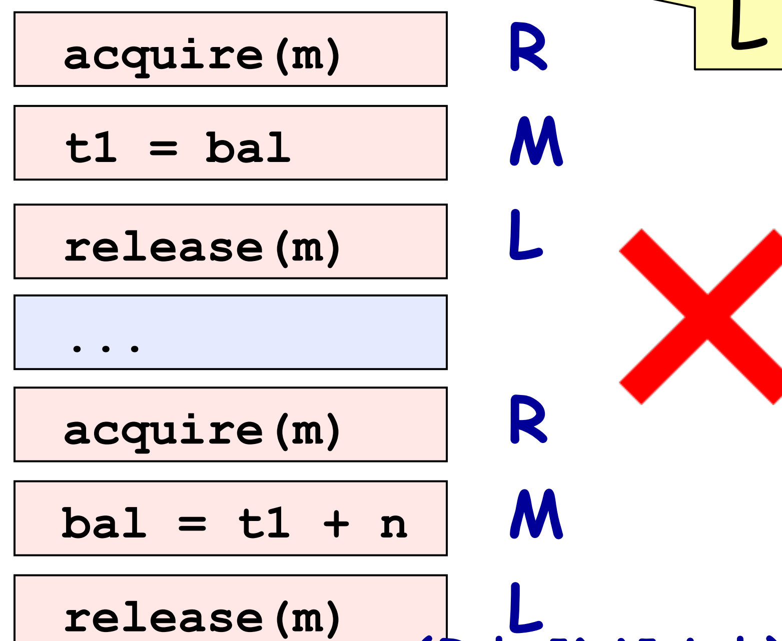
(R* [N] L*)

Examples

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




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



(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 i accesses x only when b == i

Thread A

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

Thread B

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

Execution Trace

Thread A

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

Thread B

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

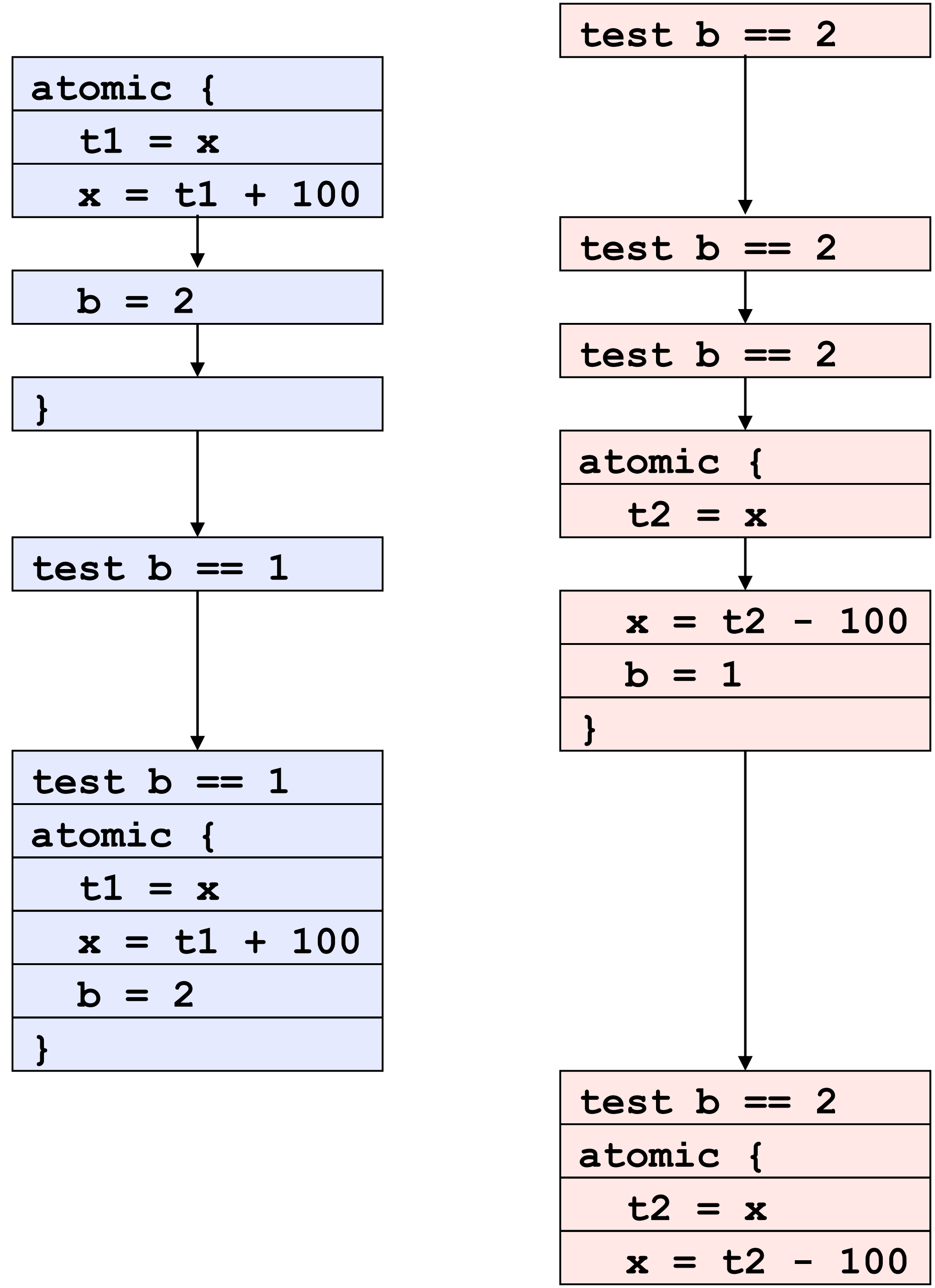
Thread A	Thread B
	test b == 2
atomic { t1 = x x = t1 + 100	
	test b == 2
b = 2	
	test b == 2
}	
	atomic { t2 = x
test b == 1	
	x = t2 - 100 b = 1 }

test b == 1
atomic {
t1 = x
x = t1 + 100
b = 2
}	

	test b == 2 atomic { t2 = x x = t2 - 100

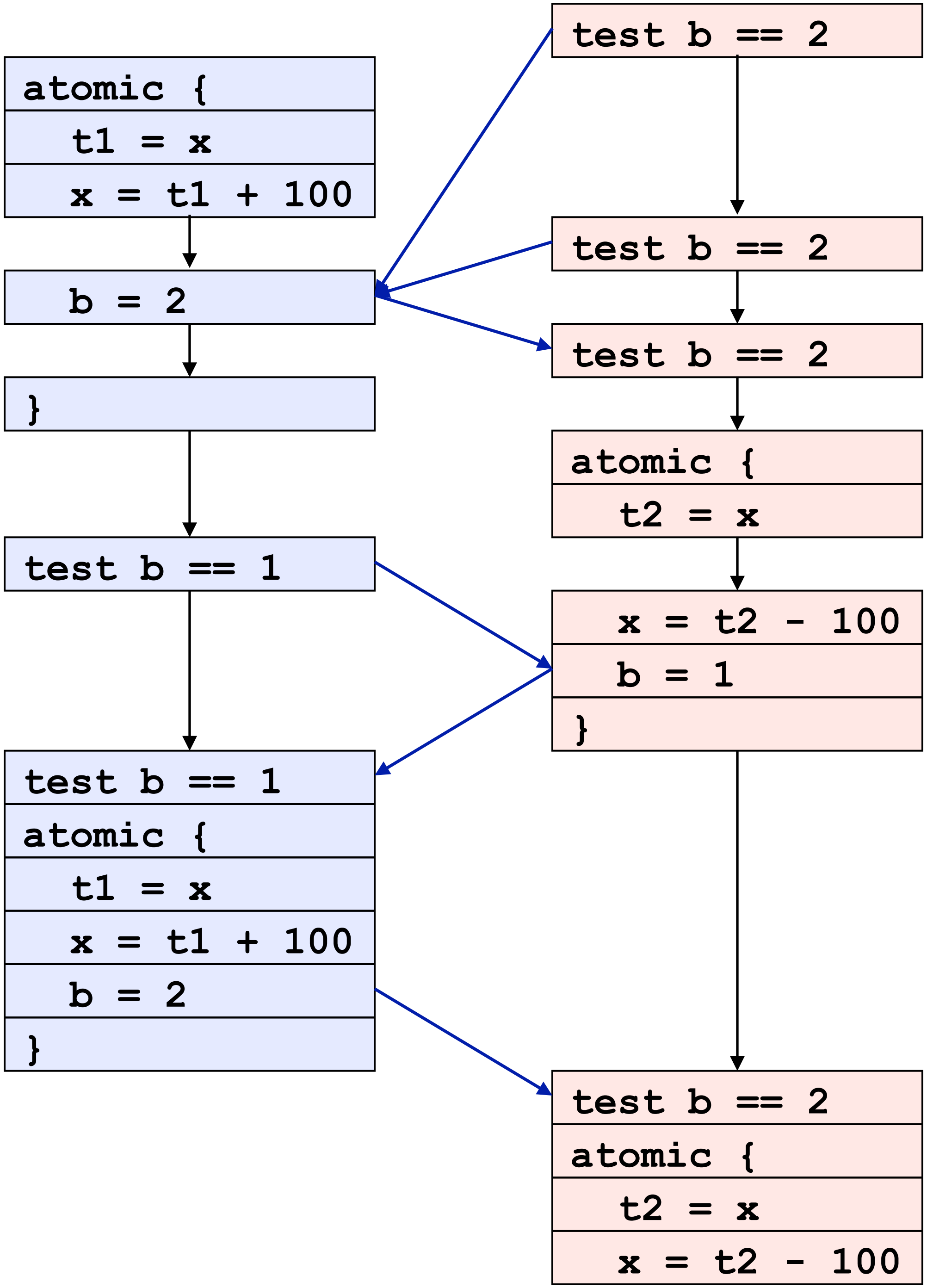
Happens-Before Ordering on Operations

- program order



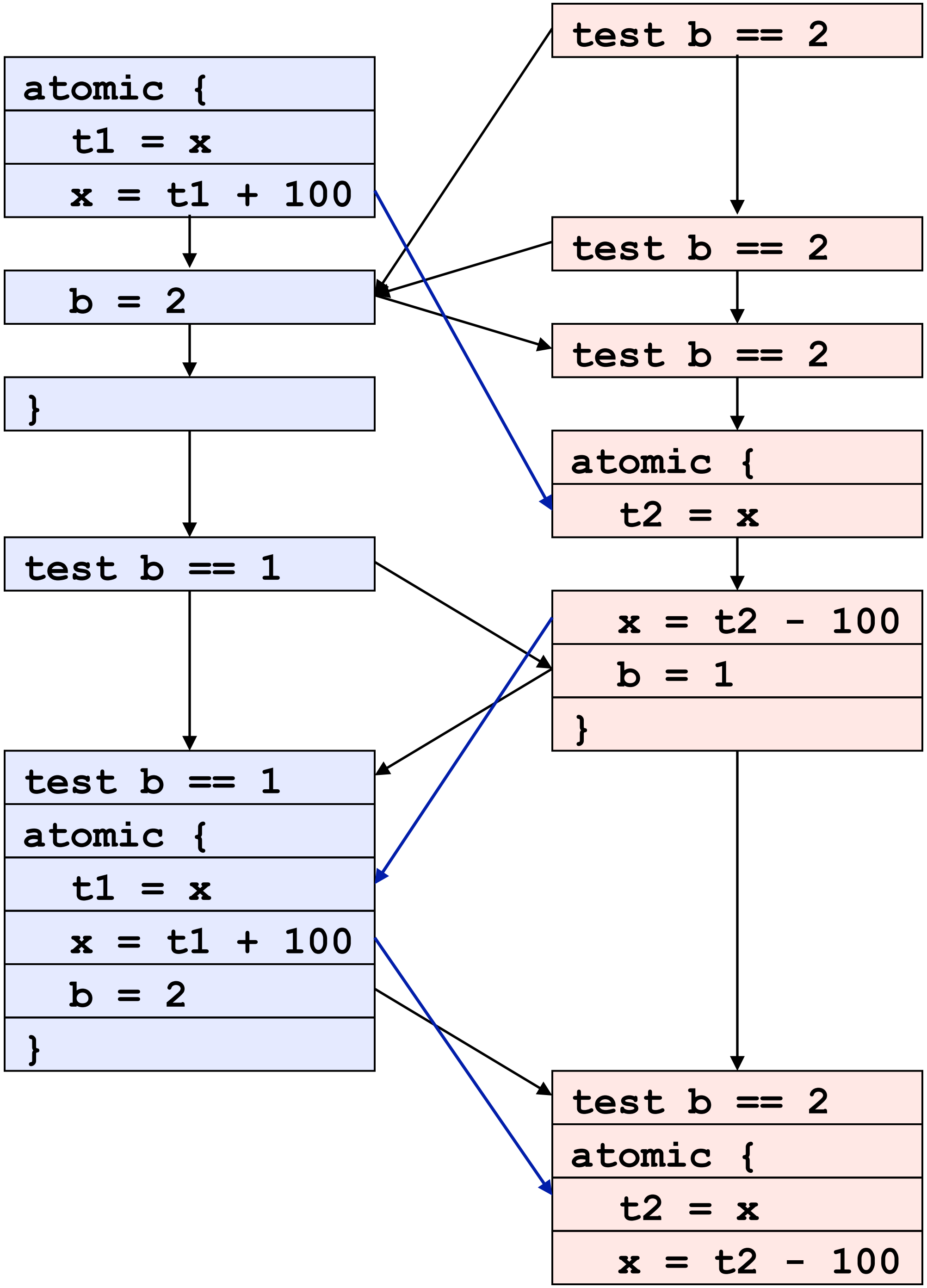
Happens-Before Ordering on Operations

- program order
- synchronization order

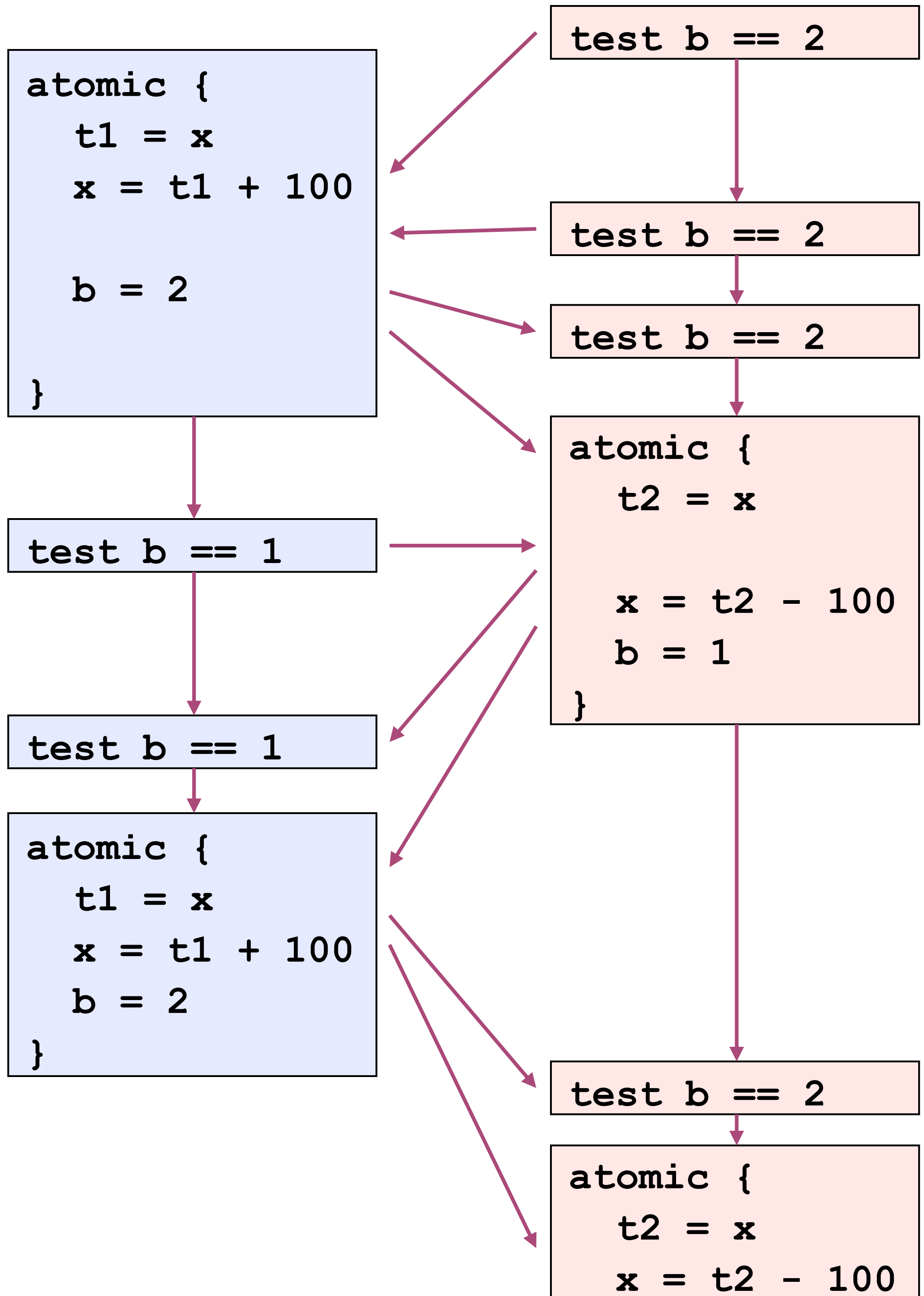


Happens-Before Ordering on Operations

- program order
- synchronization order
- communication order



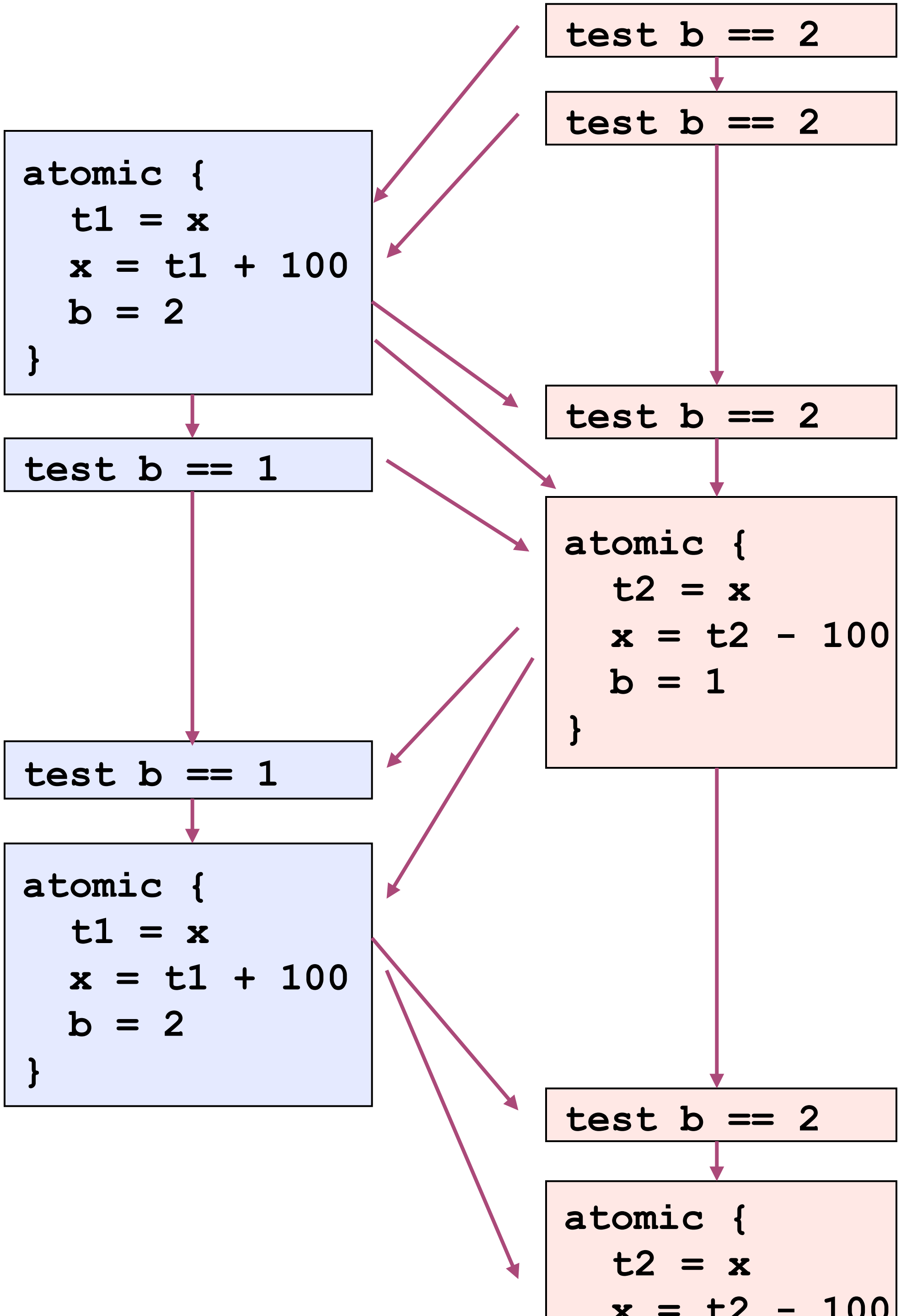
Transactional Happens-Before Ordering



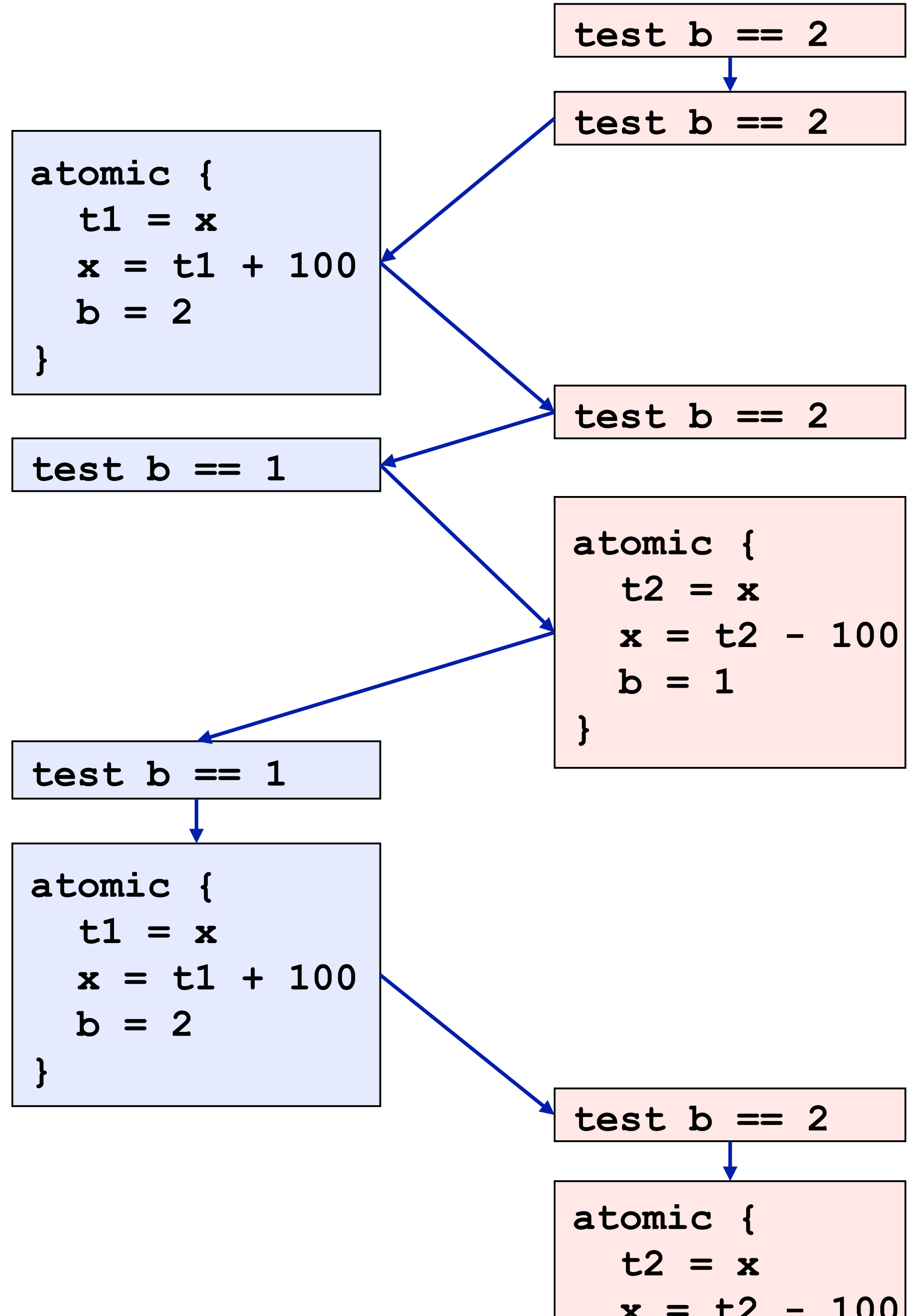
Theorem

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

Equivalent Serial Trace



Equivalent Serial Trace



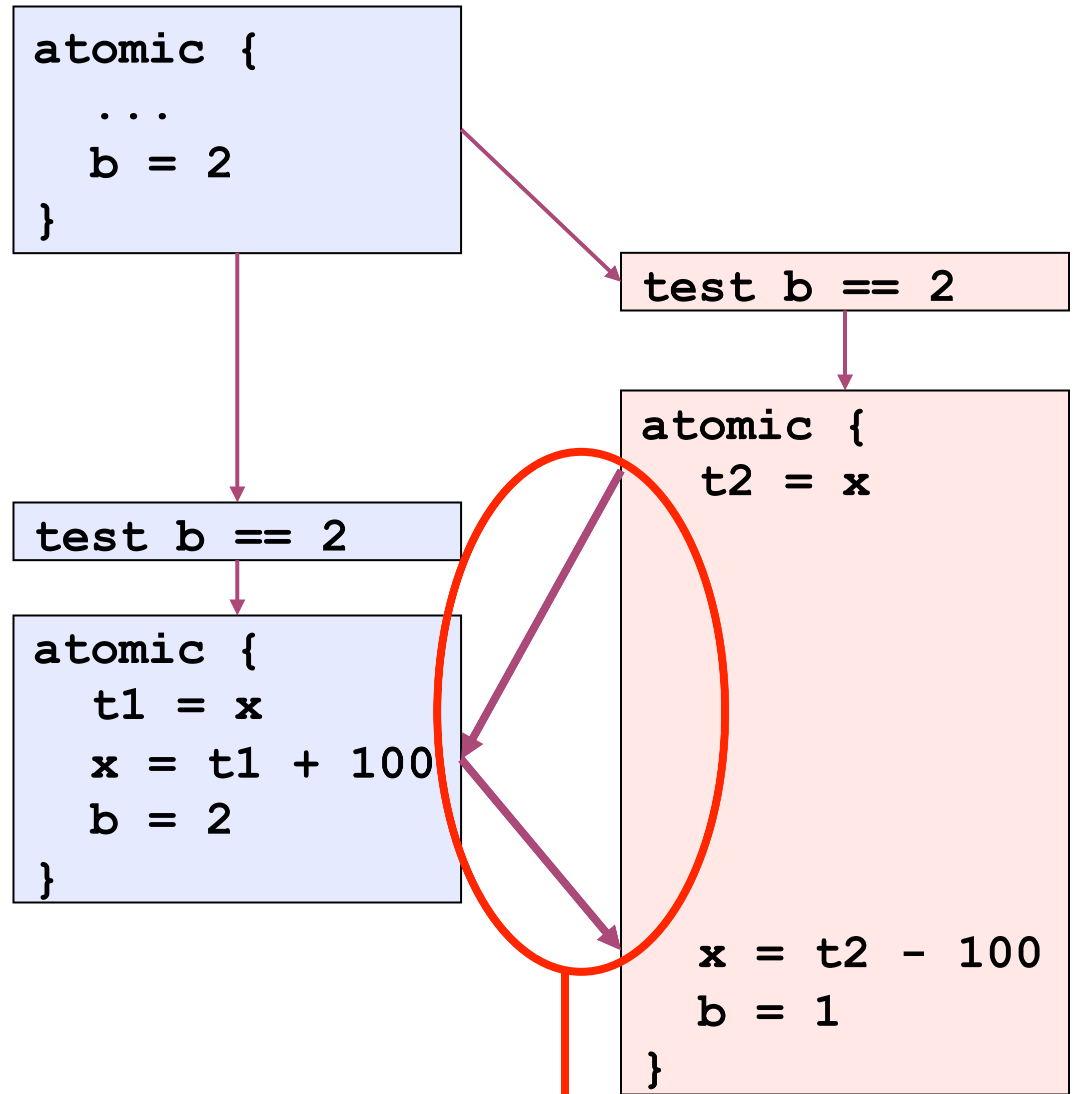
Atomicity Violation

Thread A

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

Thread B

```
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

```
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

Controlling Thread Interference: #3 Enforce Atomicity

Atomic method must behave as if it executed serially, without interleaved operations of other thread

- 10% of methods
- No information about thread interference
- Local atomic blocks awkward

```
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 method must behave as if it executed serially, without interleaved operations of other thread

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

 while (x < len) {

  tmp = a[x];

  b[x] = tmp;

  x++;

 }
}
```

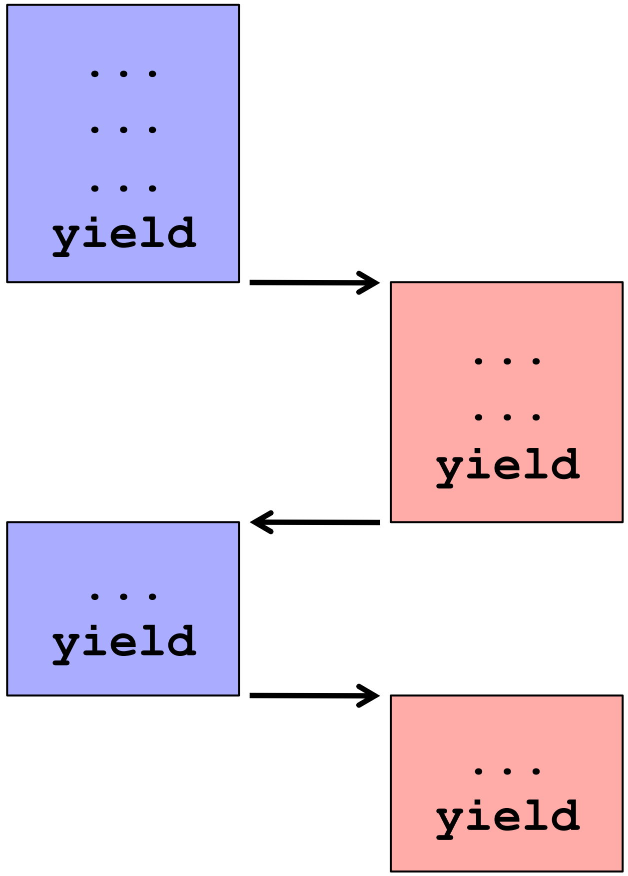
```
void busy_wait() {
 acq(m);
 thread interference?
 while (!test()) {
  thread interference?
  rel(m);
  thread interference?
  acq(m);
  thread interference?
  x++;
  thread interference?
 }
}
```

Bimodal Semantics

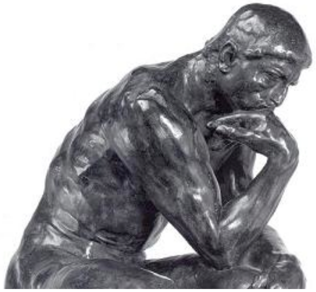
atomic
vs.
read-modify-write

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
- Limitation: Uses only a single processor



Yield-Oriented Concurrency



Cooperative Scheduler

- Sequential Reasoning
- Except at yields

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

```
...
barrier
yield
```

```
...
yield
```

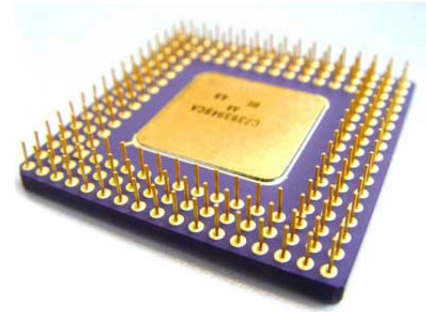
```
acq(m)
x = 2
rel(m)
yield
```

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

yields mark all interference points

Yield Correctness

Cooperative Correctness ∧ Yield Correctness → Preemptive Correctness



Preemptive Scheduler

- Full performance
- No overhead

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

```
...
barrier
yield
```

```
...
yield
```

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

Benefits of Yield over Atomic

- Atomic methods are those with no yields

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

  while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

  }
}
```

```
void busy_wait() {
  acq(m);
 thread interference?
  while (!test()) {
    thread interference?
    rel(m);
    thread interference?
    acq(m);
    thread interference?
    x++;
    thread interference?
  }
}
```

Benefits of Yield over Atomic

- Atomic methods are those with no yields

```
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);

    x++;

  }
}
```

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

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);

    x++;

  }
}
```

x++ is always an increment operation

Non-Interference Design Space

Non-Interference Specification

Policy Enforcement

	atomic	yield
traditional sync + analysis	atomicity, serializability	**yield-oriented programming**
new run-time systems	transactional memory	automatic mutual exclusion

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

Multiple Threads

x++

is a non-atomic read-modify-write

```
x = 0;

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++;

}
```

Yield-Oriented Programming

x++ vs.

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

```
x = 0;

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

    x++;

}
```

Single Thread

x++

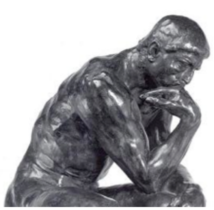
```
x = 0;

while (x < len) {

    tmp = a[x];

    b[x] = tmp;

    x++;

}
```

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() {...}
```

Version 1

```
volatile int x;

void update_x() {

  x = slow_f(x);

}
```



No yield between accesses to x

Cooperative Correctness ∧ Coop/preemptive Equivalence → Preemptive Correctness

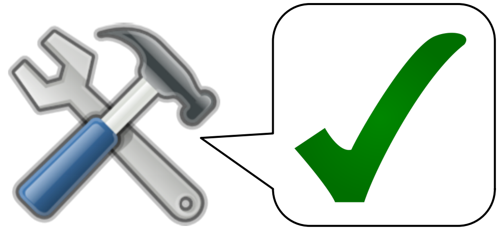
Version 2

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

But...
Bad performance

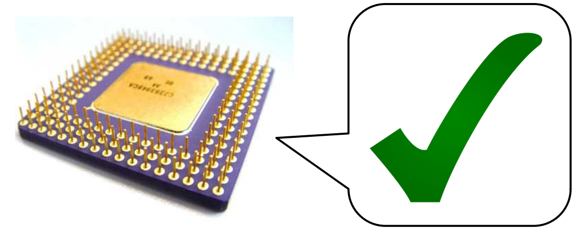
Cooperative Correctness

∧



Coop/preemptive Equivalence

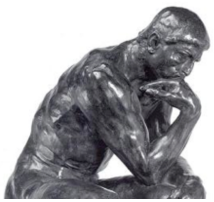
➔



Preemptive Correctness

Version 3

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

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

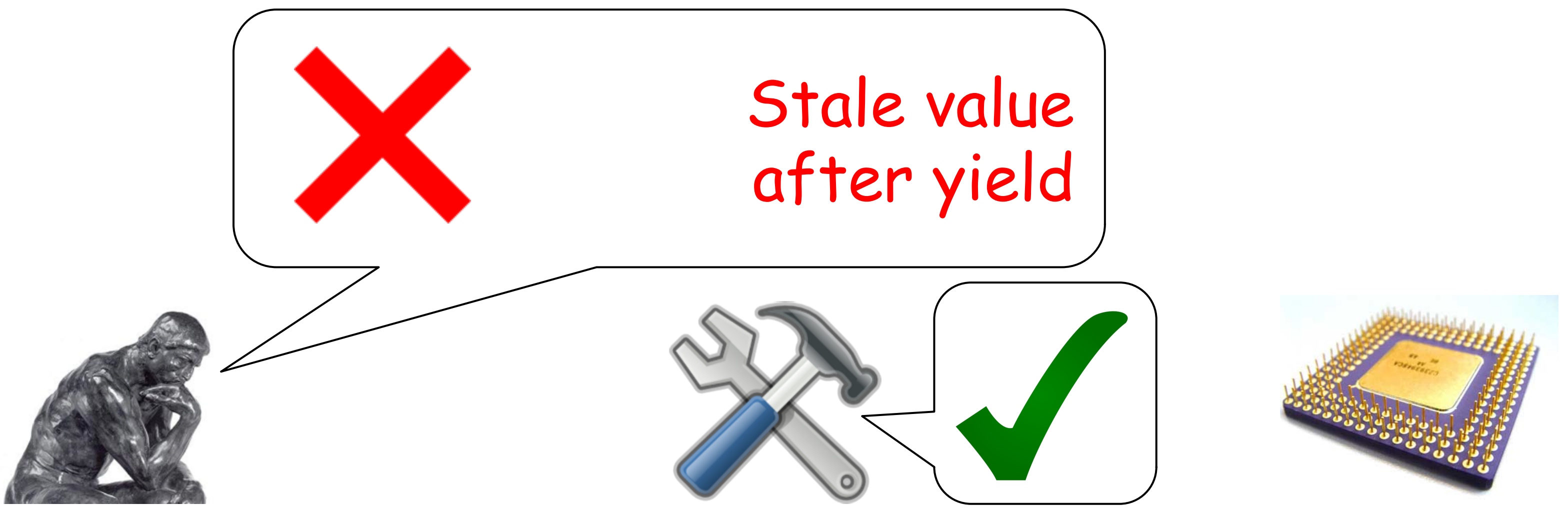


No yield between accesses to x

Cooperative Correctness ∧ Coop/preemptive Equivalence ➔ Preemptive Correctness

Version 4

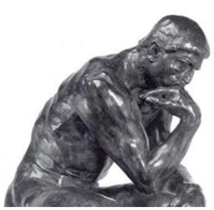
```
void update_x() {
  int fx = slow_f(x);
  yield;
  acquire(m);
  x = fx;
  release(m);
}
```



Cooperative Correctness $\wedge$ Coop/preemptive Equivalence $\rightarrow$ Preemptive Correctness

Version 5
(test and retry)

```
void update_x() {
  int y = x;
  for (;;) {
    yield;
    int fy = slow_f(y);

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

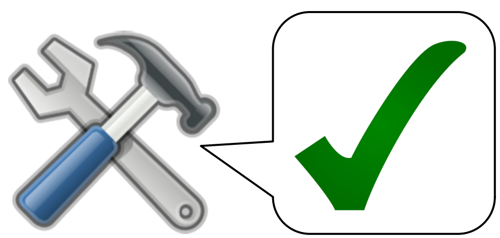


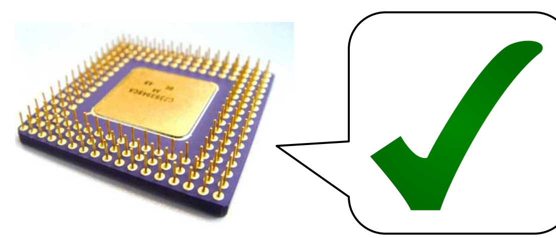
No yield between accesses to x

Cooperative Correctness $\wedge$ Coop/preemptive Equivalence ➔ Preemptive Correctness

Version 6

```
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);
  }
}
```





Cooperative Correctness $\wedge$ Coop/preemptive Equivalence $\rightarrow$ Preemptive Correctness

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?

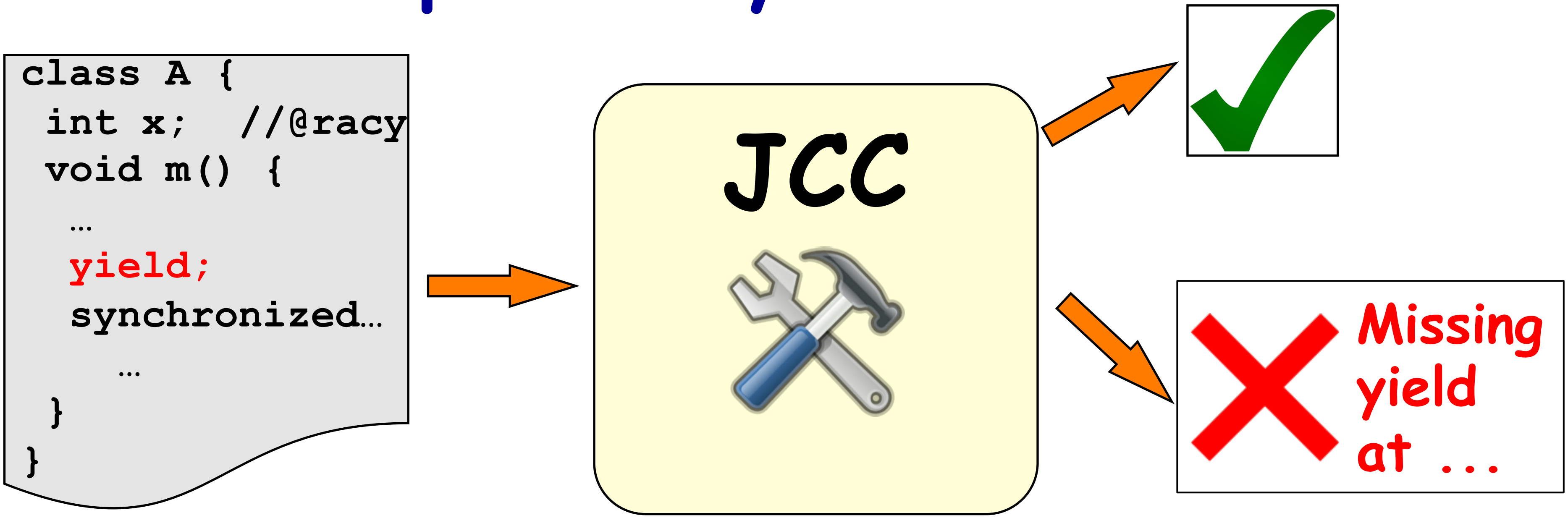
StringBuffer	Concurrency bug	Some other bug	Didn't find bug	Total
Yields	10	1	1	12
No Yields	1	5	9	15

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

Static Program Analysis for Cooperability

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 Yield-Correct Code

- Commuting Classifications

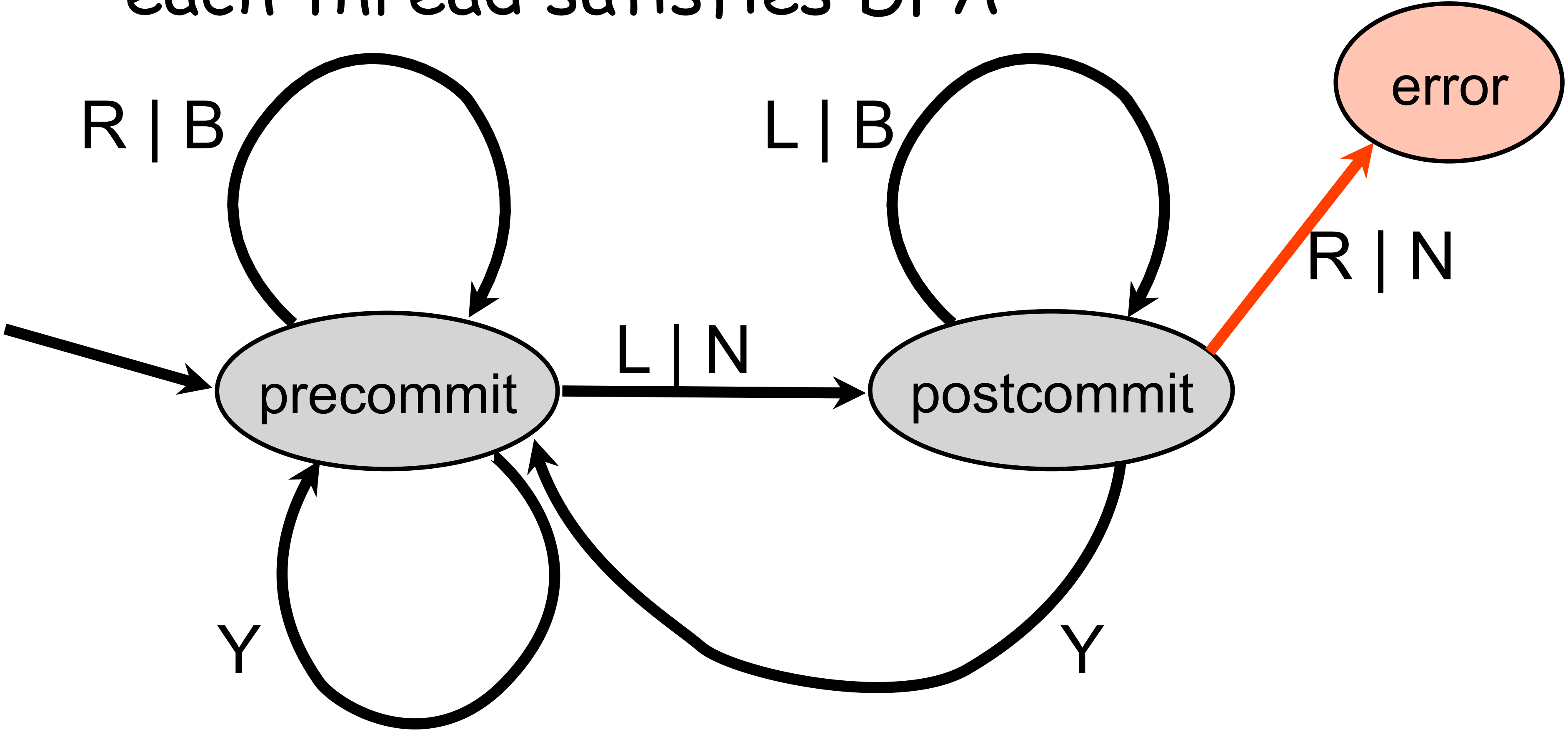
R	Right-mover	Acquire
L	Left-mover	Release
M	Both-mover	Race-Free Access
N	Non-mover	Racy Access
Y	Yielding	`yield`

- Cooperable blocks have the pattern:

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

Coop/Preemptive Equivalence

- Trace is coop/preemptive equivalent if each thread satisfies DFA



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

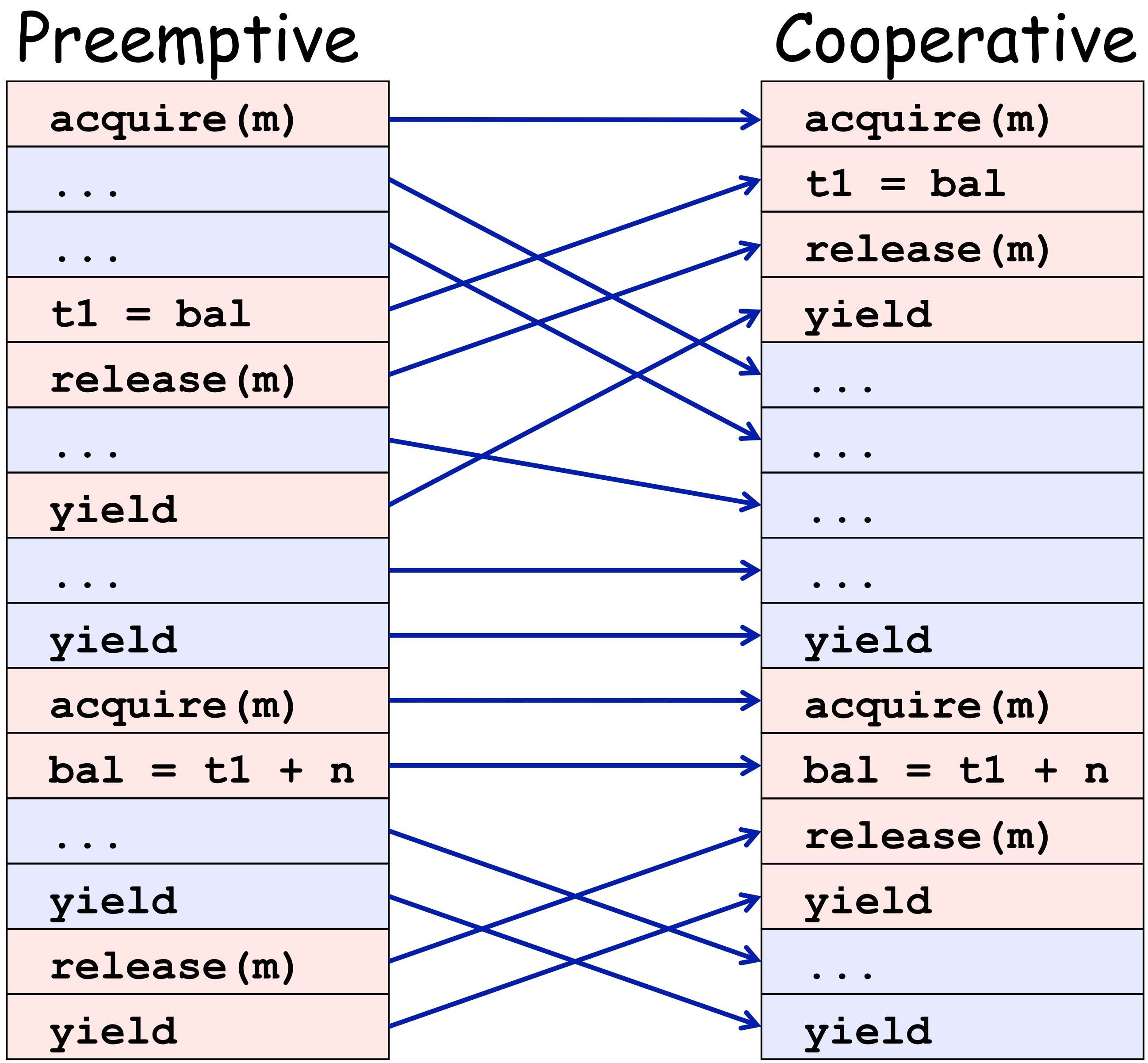
Examples

```
void deposit(int n) {
  synchronized(m) {      // R
    t1 = bal;            // M
  }                      // L
  yield;                 // Y
  synchronized(m) {      // R
    bal = t1 + n;        // M
  }                      // L
}
```

acquire(m)	R
t1 = bal	M
release(m)	L
yield	Y
...	
acquire(m)	R
bal = t1 + n	M
release(m)	L

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

Traces



Preemptive
acquire(m)
...
...
t1 = bal
release(m)
...
yield
...
yield
acquire(m)
bal = t1 + n
...
yield
release(m)
yield
Cooperative
acquire(m)
t1 = bal
release(m)
yield
...
...
...
...
yield
acquire(m)
bal = t1 + n
release(m)
yield
...
yield

Summary of Static Analysis

- Compute an *effect* for each program expression/statement
- Effect summarizes how that computation interacts with other threads

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

```
class TSP {

  volatile int shortestPathLength;

  void searchFrom(Path path) {
    if (path.length() >= shortestPathLength) return;

    if (path.isComplete()) {


        if (path.length() < shortestPathLength)
          shortestPathLength = path.length();

    } else {
      for (Path c : path.children()) {

        searchFrom(c);
      }
    }
  }
}
```

Racy Read

Non-Racy Read

Racy Write

```
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 shortest

  void searchFrom(Path
    if (path.length() >

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

```
class Path {
  mover int length() ...
  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);
      }
    }
  }
}
```

series of transactions that do not commute

```
class TSP {
  Object lock;
  volatile int shortestPathLength; // lock held on writes

  compound void searchFrom(Path path) {
    if (path.length() >= shortestPathLength) return;

    if (path.isComple    ) {
      yield;
      synchronized(lock) {
        if (path.length() < shortestPathLength)
          shortestPathLength = path.length();
      }
    } else {
      for (Path c : path.children()) {
        yield;
        searchFrom(c);
      }
    }
  }
}
```

M

N

((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);
      }
    }
  }
}
```

M; N

M

Y

R

M; M

M; N

L

((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);
      }
    }
  }
}
```

M; N

M

M

Y

N

(Y;N)*

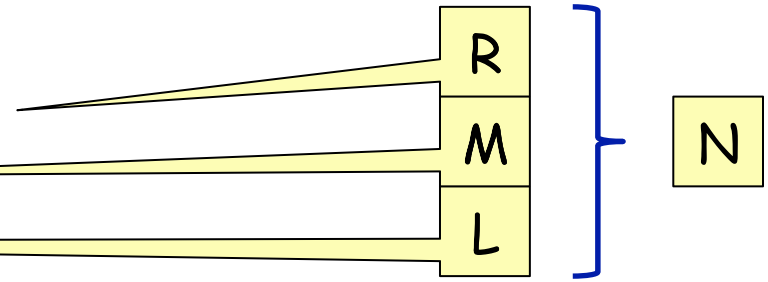
((R* [N] L*) Y)* [R* [N] L*]

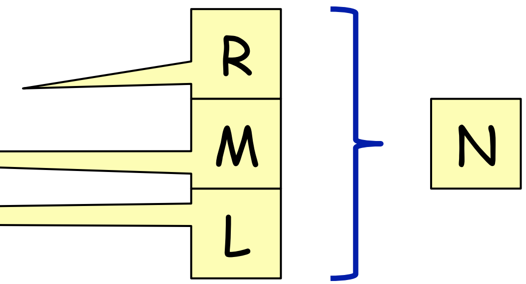
```
class TSP {
  Object lock;
  volatile int shortestPathLength; // lock held on writes

  compound void searchFrom(Path path) {
    if (path.length() >= shortestPathLength) return;        M; N

    if (path.isComplete()) {                                 M
      yield;                                                 Y

        if (path.length() < shortestPathLength)              M; N
          shortestPathLength = path.length();                M; N
    } else {
      for (Path c : path.children()) {
        yield;
        searchFrom(c);
      }
    }
  }
}
```

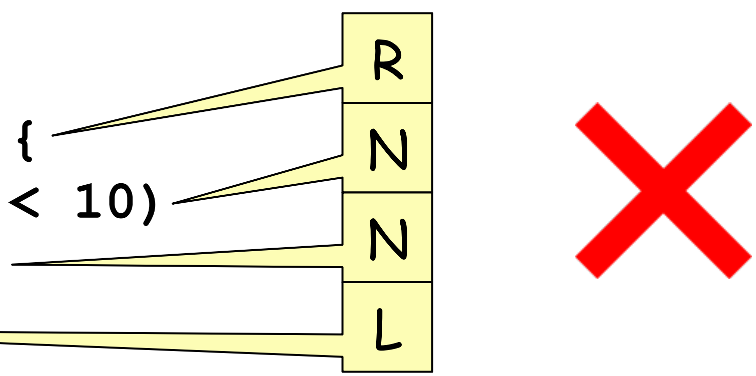
((R* [N] L*) Y)* [R* [N] L*]

Conditional Effects

```
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;

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

  this ? mover : non-mover
  synchronized void add(String s) {
    ...
  }
}
```

	this not held	this held
synchronized int length() {	R	M
return count;	M	M
}	L	M
synchronized void add(String s) {	R	M
...	M	M
}	L	M

```
StringBuffer sb;
synchronized (sb) {
  if (sb.length() < 10)
    sb.add("moo");
}
```

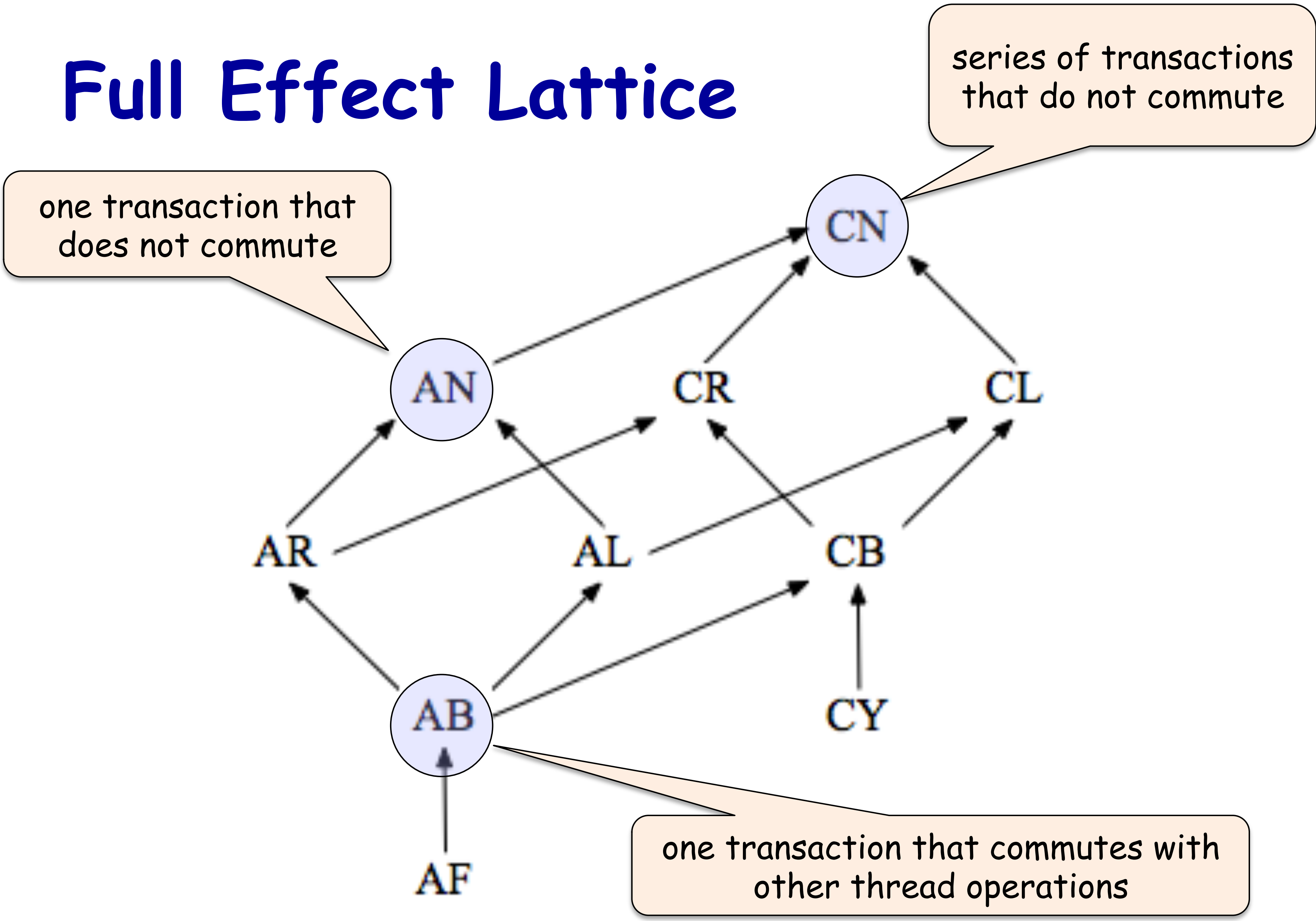
synchronized (sb) {	R
if (sb.length() < 10)	M
sb.add("moo");	M
}	L

✔

```
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);
      }
    }
  }
}
```

Full Effect Lattice



Program	Size (LOC)	Annotation Time (min.)	Anotation Count
java.util.zip.Inflater	317	9	4
java.util.zip.Deflater	381	7	8
java.lang.StringBuffer	1,276	20	10
java.lang.String	2,307	15	5
java.io.PrintWriter	534	40	109
java.util.Vector	1,019	25	43
java.util.zip.ZipFile	490	30	62
sparse	868	15	19
tsp	706	10	45
elevator	1,447	30	64
raytracer-fixed	1,915	10	50
sor-fixed	958	10	32
moldyn-fixed	1,352	10	39
Total	13,570	231	490
Total per KLOC		17	36

Program	Number of Interference Points					Unintended Yields
	No Spec	Race	Atomic	Atomic Race	Yield	
java.util.zip.Inflater	36	[illegible]	[illegible]	[illegible]	0	0
java[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0
jav[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1
jav[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0
jav[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	9
jav[illegible]	[illegible]	106	44	24	4	1
jav[illegible]	[illegible]	105	85	53	30	0
spar[illegible]	[illegible]	98	48	14	6	0
tsp	445	115	437	80	19	0
elevator	454	[illegible]	[illegible]	[illegible]	[illegible]	0
raytracer-fixed	565	[illegible]	[illegible]	[illegible]	[illegible]	2
sor-fixed	249	[illegible]	[illegible]	[illegible]	[illegible]	0
moldyn-fixed	983	130	[illegible]	37	30	0
Total	3,928	1,291	1,890	432	180	13
Total per KLOC	289	95	139	32	13	1

Interference [illegible]

- *racy* field ac[illegible]
- all lock acquires
- atomic method calls

in non-atomic methods

Interference at:

- *yield points*

[illegible]erence at:

- [illegible] field accesses
- all lock acquires

Fewer Interference Points: Easier to Reason about Code!

Dynamic Program Analysis for Cooperability

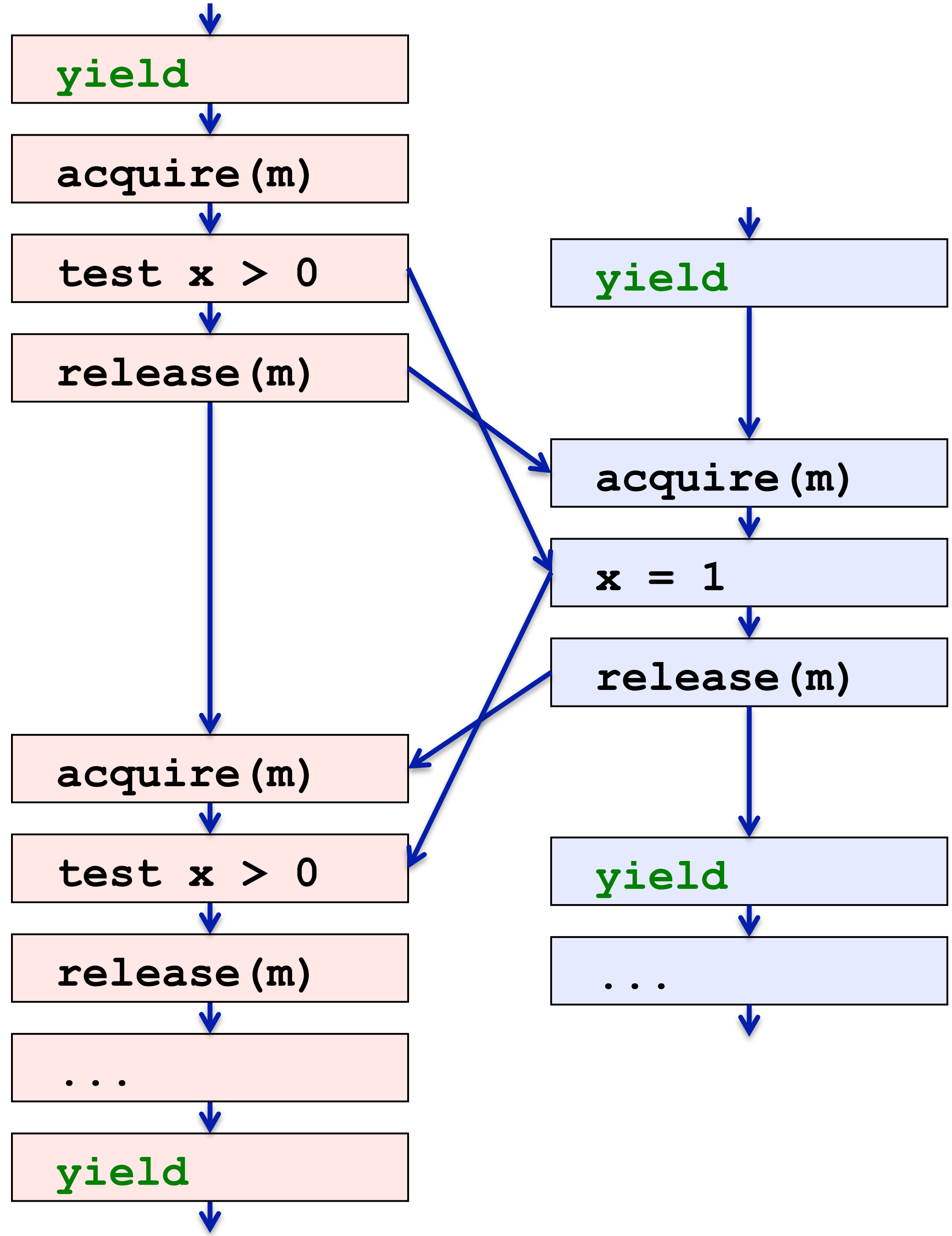
Copper [PPOPP 11]

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

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

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

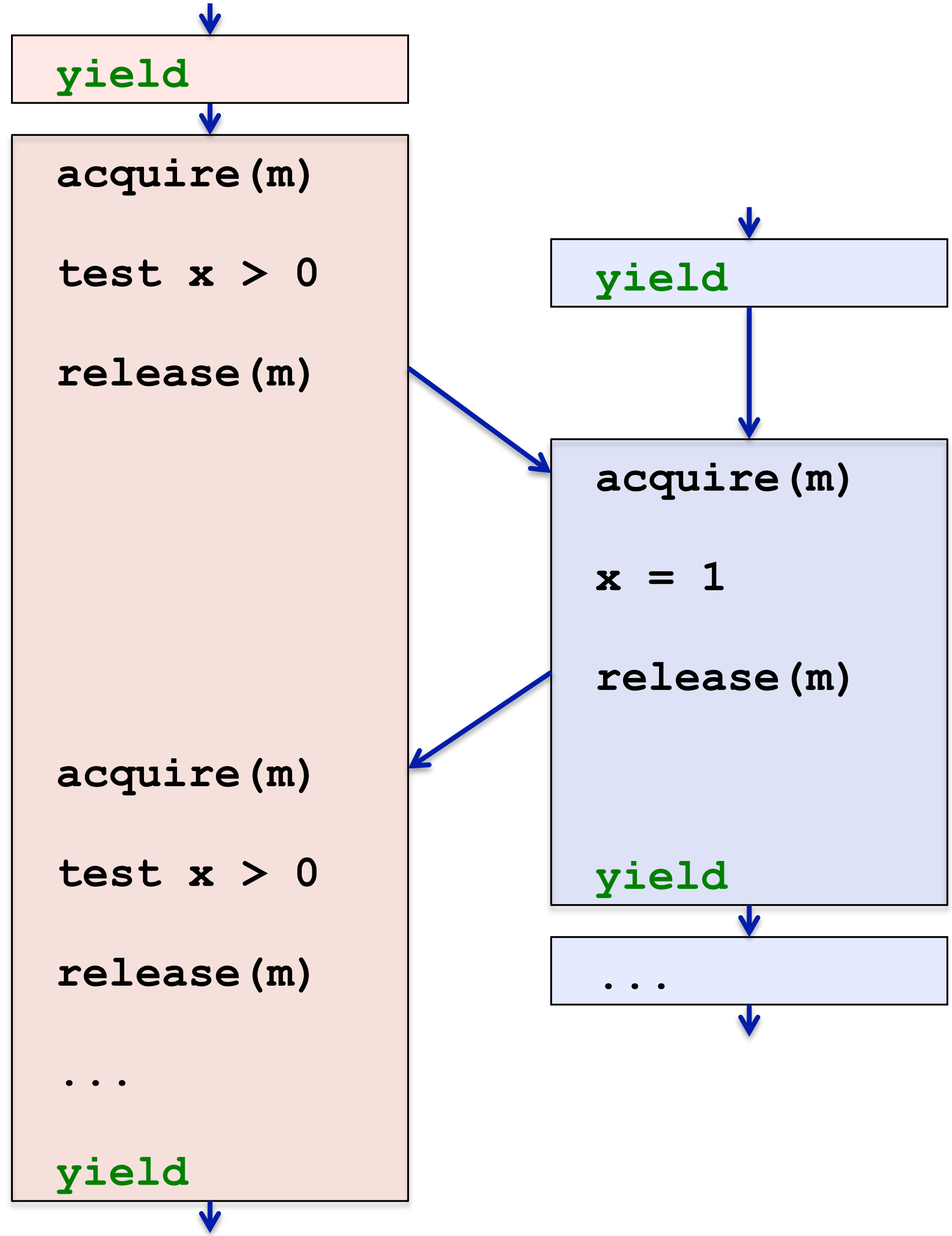
Copper

- Build Transactional Happens-Before
  - program order
  - sync. order
  - comm. order



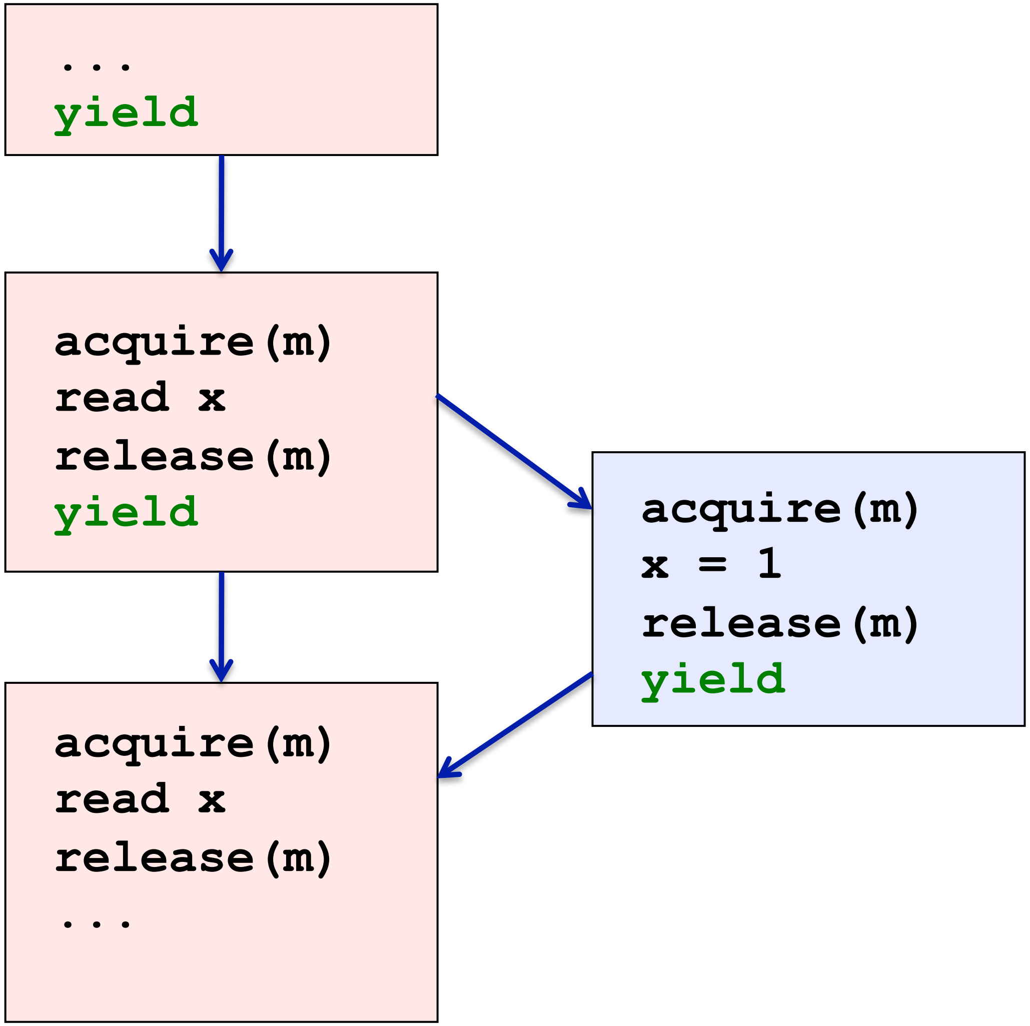
Copper

- Build Transactional Happens-Before
- Yields mark transaction ends
- Cycles indicate missing yields



Copper

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



Copper Results

program	LLOC	No Analysis	Atomic Methods	Yields
sparse	712			0
sor				
seri				
cry				
mo			64	
ele			54	3
lufa			57	3
rayt			65	3
montecarlo	3557	377	41	1
hedc	6409	305		
mtrt	6460	695		
raja	6863	396	45	0
colt	25644	601	113	13
jigsaw	48674	3415	550	47

Interference at:

- field accesses
- all lock acquires
- atomic method calls

in non-atomic methods

Interference at:

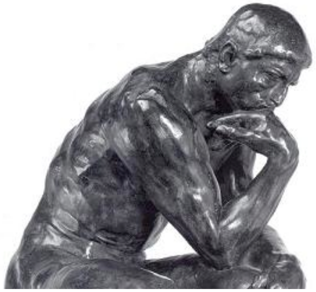
- *yield points*

...ence at:

- ...d accesses
- all lock acquires

Fewer interference points: less to reason about!

Cooperative Concurrency



Cooperative Scheduler
- Sequential Reasoning
- Except at yields

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

```
...
barrier
yield
```

```
...
yield
```

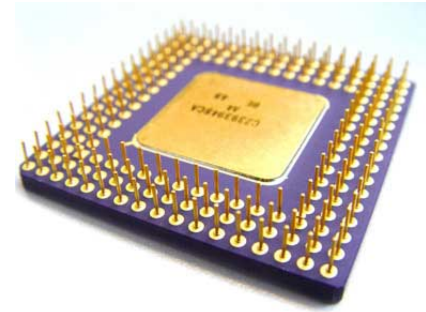
```
acq(m)
x = 2
rel(m)
yield
```

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

yields mark all thread interference

Yield Correctness

$\cong$



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 $\wedge$ Yield Correctness $\cong$ $\rightarrow$ Preemptive Correctness

Summary

- Race freedom
  - code behaves **as if** on sequentially consistent memory model
  - http://www.cs.williams.edu/~freund/papers.html
- Atomicity
  - code behaves **as if** atomic methods executed serially
  - http://users.soe.ucsc.edu/~cormac/atom.html
- 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

Where To Go From Here?

- Static Race Checking Analysis
- Performance (goal is always-on precise detection...)
  - HW support
  - static-dynamic hybrid analyses
  - sampling
- Coverage
  - symbolic model checking, specialized schedulers
- Classify malignant/benign data races
  - which data races are most critical?
- How to respond to data races? warn/fail-fast/recover?
- Reproducing traces exhibiting rare data races
  - record and replay
- Generalization: reason about traces beyond the observed trace

Where To Go From Here?

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

Key References

- R. J. Lipton. Reduction: A method of proving properties of parallel programs. CACM, 1975.
- C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. TOPLAS, 30(4), 2008.
- C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. POPL, pages 256–267, 2004.
- C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. PLDI, pages 293–303, 2008.
- J. R. Larus and R. Rajwar. Transactional Memory, 2006.
- M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. ISCA, pages 289–300, 1993.

Key References

- M. Isard and A. Birrell. Automatic mutual exclusion. In Workshop on Hot Topics in Operating Systems, pages 1–6, 2007.
- C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of Cooperability. In Onward! Workshop on Evaluation and Usability of Programming Languages and Tools, 2010.
- C. A. Stone, M. E. O'Neill, and The OCM Team. Observationally cooperative multithreading. In OOPSLA Companion, pages 205–206, 2011.
- L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. IEEE Transactions on Software Engineering, 32:93–110, 2006.
- J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in Java. ISSTA, pages 232–242, 2012.
- J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. PPOPP, pages 147–156, 2011.