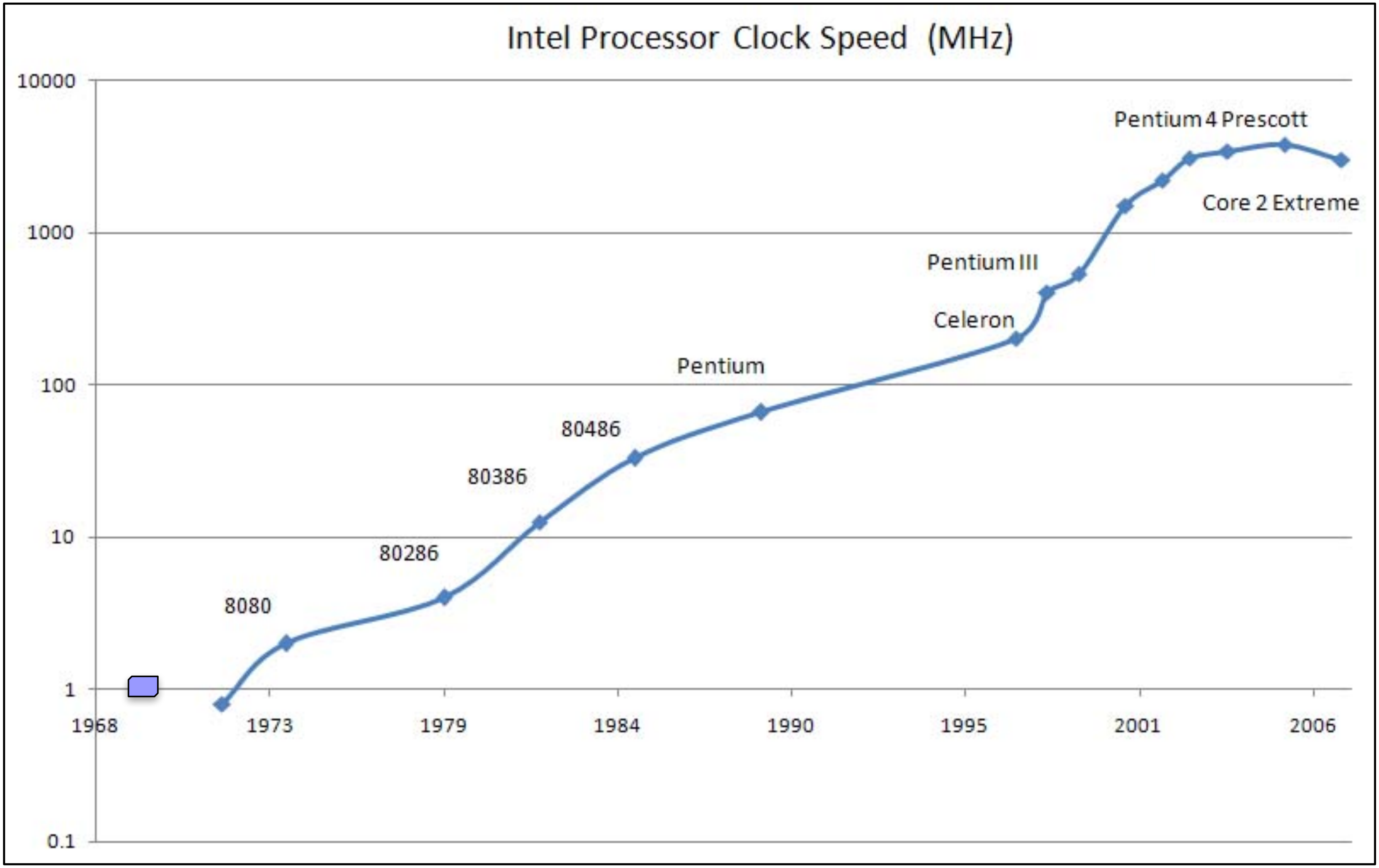
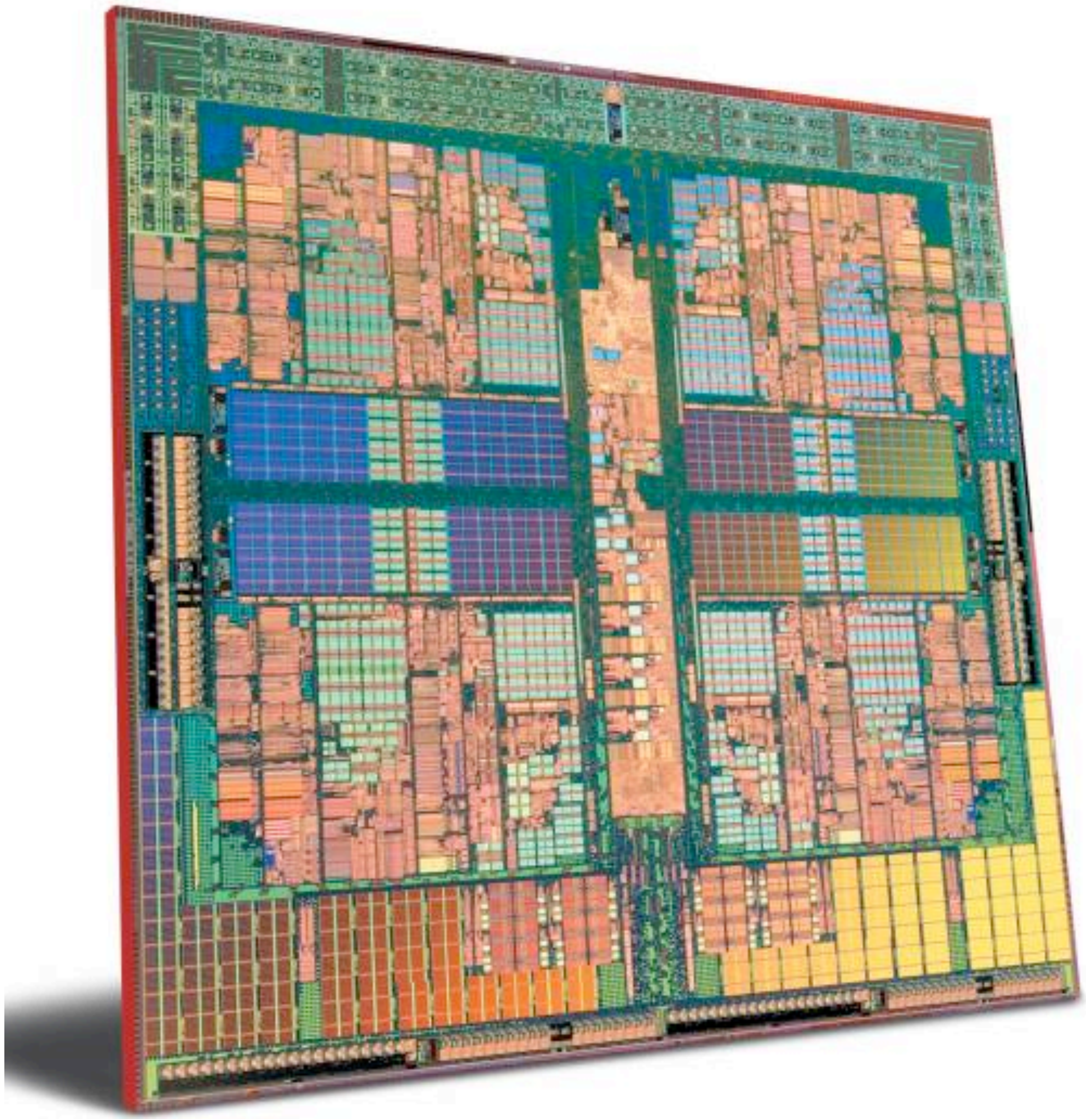
Towards Efficient and Precise Concurrent Software Analysis

Cormac Flanagan
UC Santa Cruz

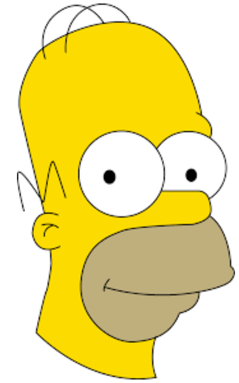
- Stephen Freund, Williams College
- Jaeheon Yi, UC Santa Cruz (now at Google)
- Caitlin Sadowski, UC Santa Cruz (now at Google)
- Tom Austin, UC Santa Cruz (now at San Jose State University)
- Tim Disney, UC Santa Cruz (now at Google)
- Dustin Rhodes (now at Google)
- Ben Wood, Williams College (now 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


Intel Processor Clock Speed (MHz)
10000
1000
100
10
1
0.1
1968
1973
1979
1984
1990
1995
2001
2006
8080
80286
80386
80486
Pentium
Celeron
Pentium III
Pentium 4 Prescott
Core 2 Extreme

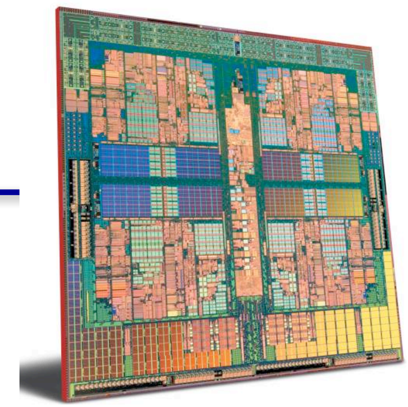
Multicore CPUs



Natural language



Programming language



Syntax

- …

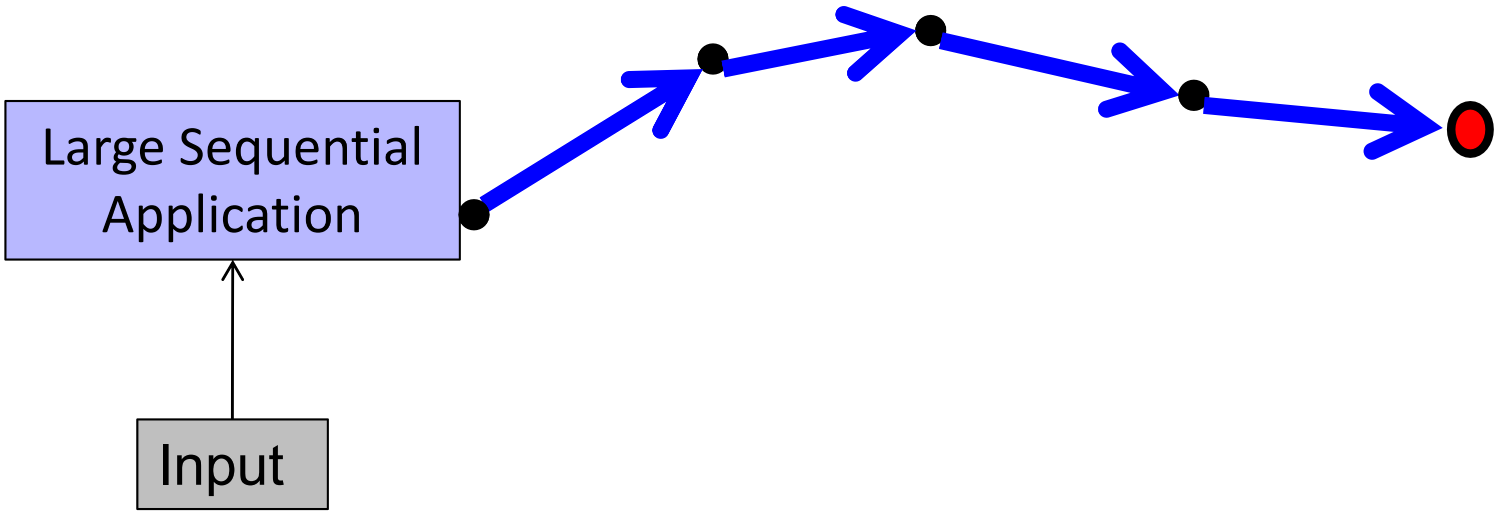
Semantics

- correctness
- modularity
- security
- testability
- ...

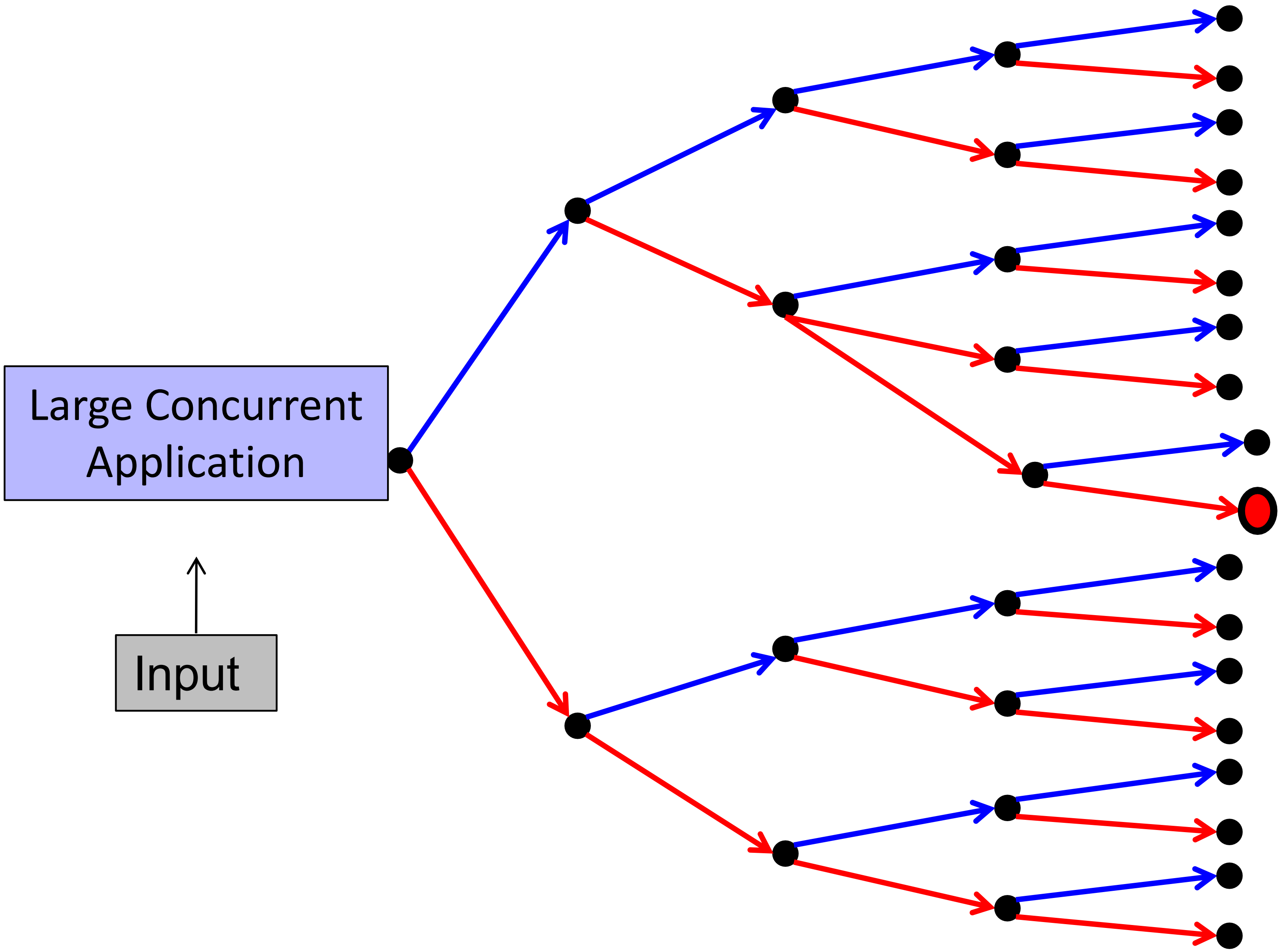
Multicore hardware

- threads
- shared memory
- preemptive scheduling
- relaxed memory models

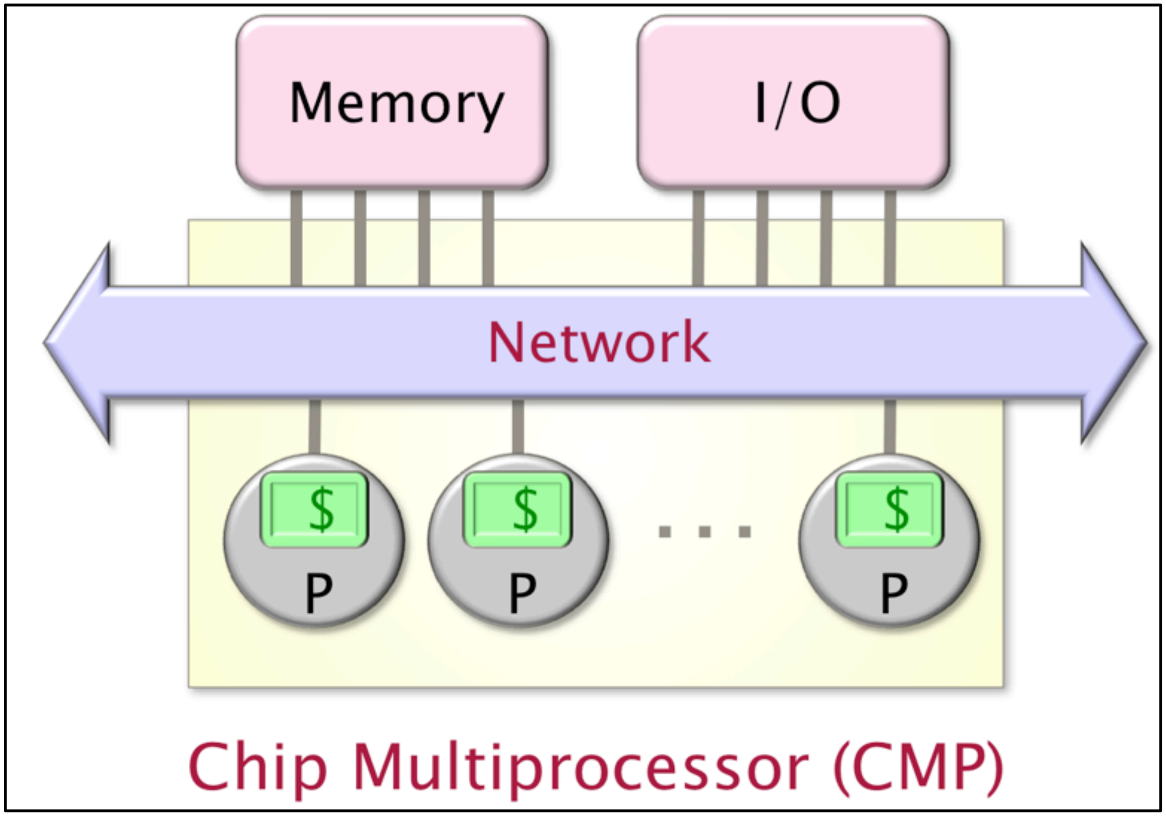
Sequential Software: Deterministic



Multithreaded Software: Nondeterministic Preemptive Scheduling



Relaxed Memory Models

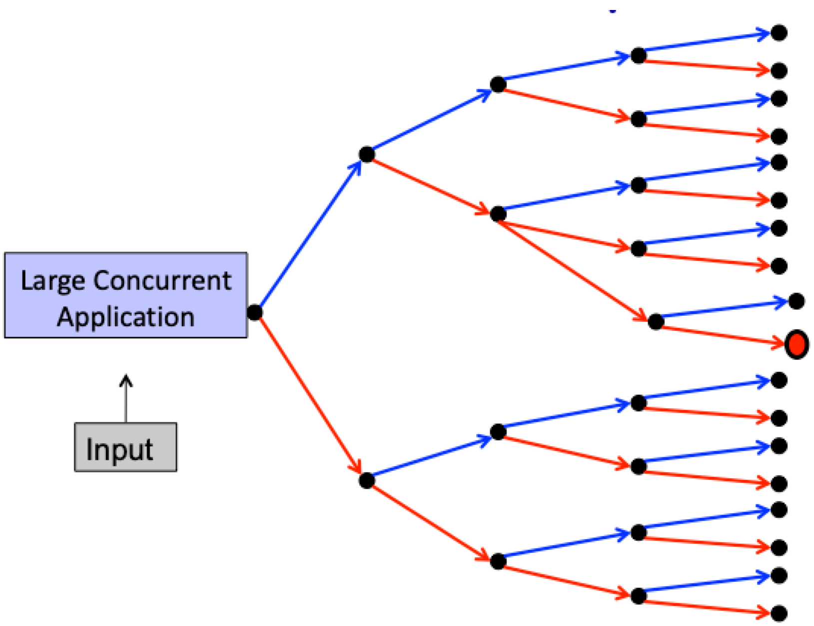


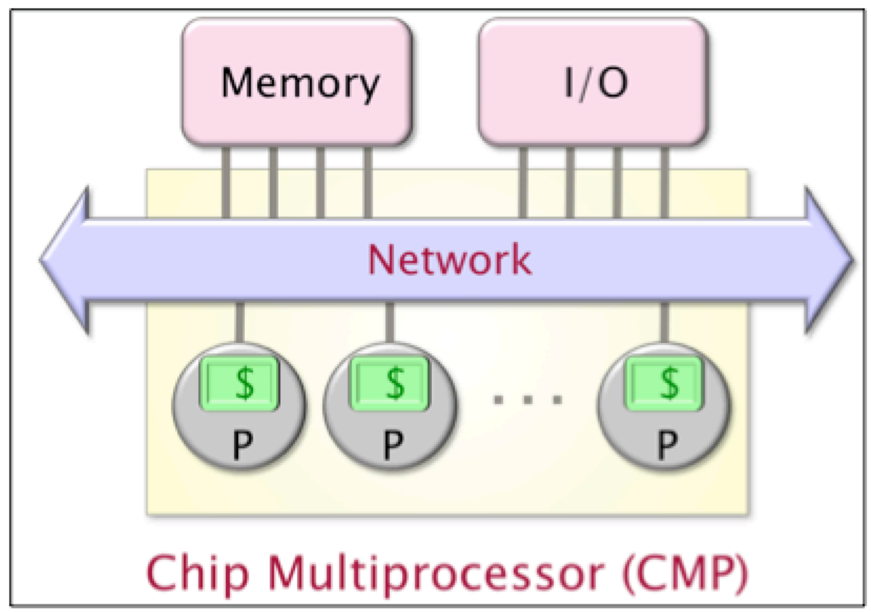
- Does each read see the "most recent" write?
  - Sequentially Consistent MM => Yes
  - Relaxed MM (JMM, x86-TSO, etc.) => No

Double NonDeterministic "Demons" of Multithreading

Preemptive scheduling

Relaxed memory model





```
x++
```

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 Enforce Race Freedom

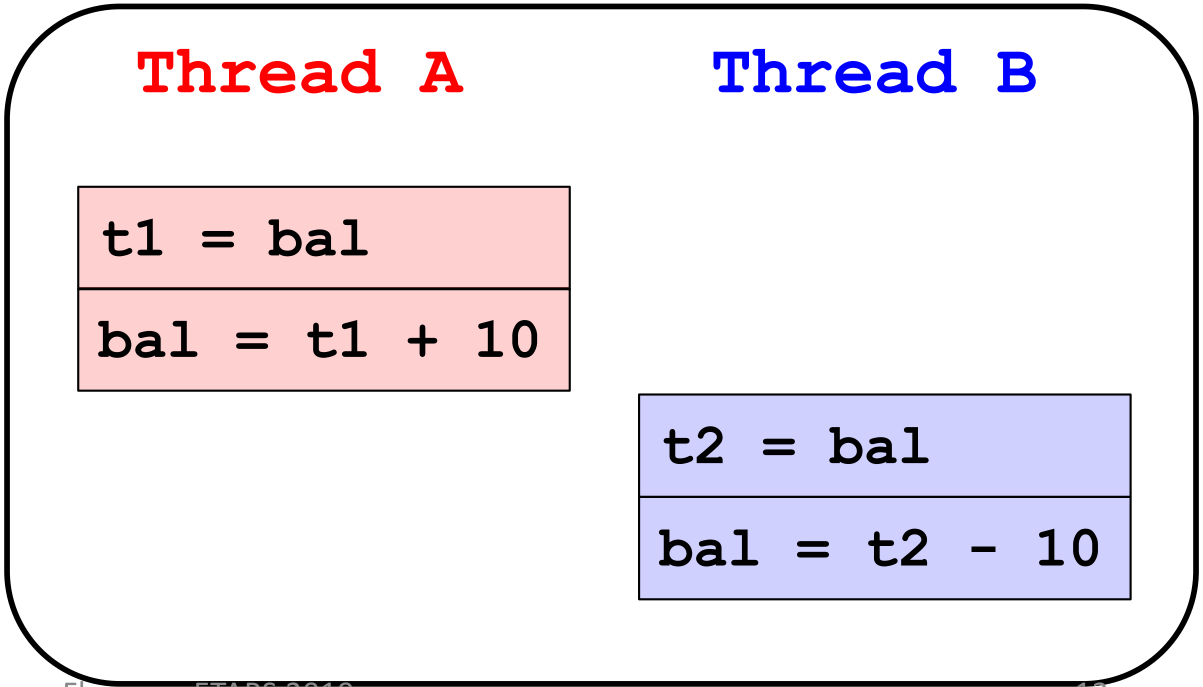
Controlling Thread Interference: #1 Enforce Race Freedom

- Race Condition

  two concurrent unsynchronized accesses, at least one write

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

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



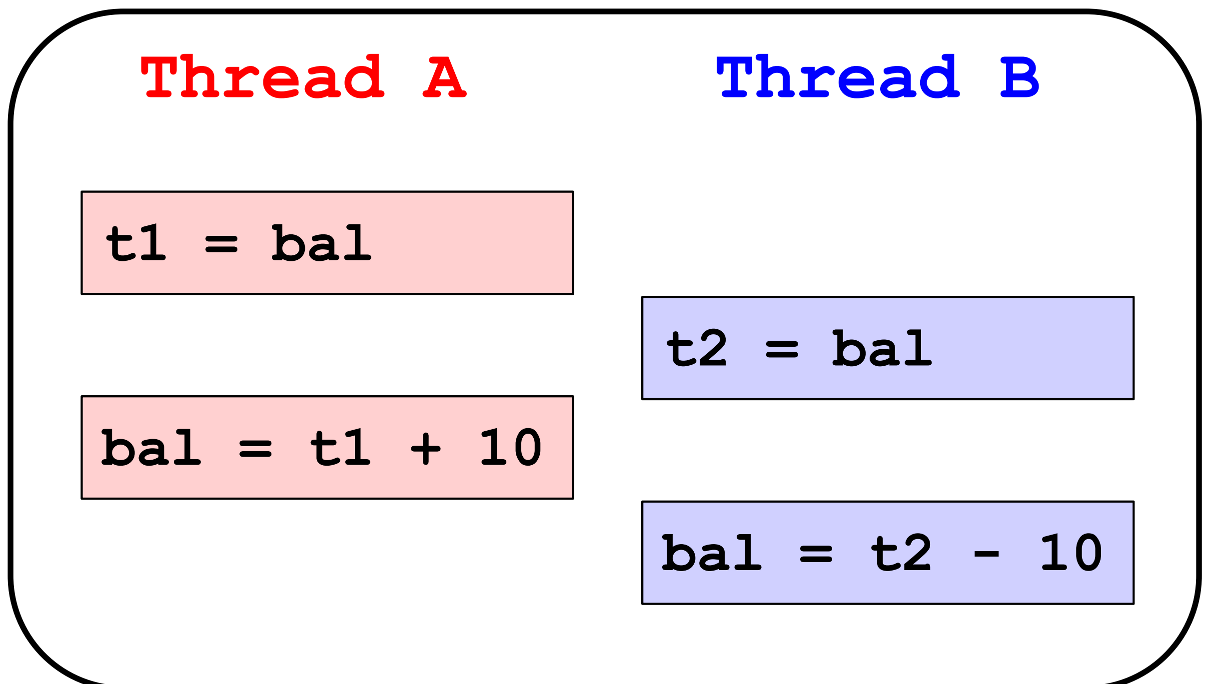
Controlling Thread Interference: #1 Enforce Race Freedom

- Race Condition

  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: #1 Enforce Race Freedom

- Many analyses to detect races
  - AAF'06, AS'04, AG'98, BR'01, DC'94, EA'03, G'03, NAW'06, VJL'07, PFH'06, PS'07, SBNSA'97, vPG'01, YRC'05, FF'09, CC'03, BCM'10
- Races are correlated to defects
- Theorem 1
  - Any race-free program behaves *as if* running on sequentially consistent memory model

Types For Race Freedom: java.util.Vector

```
class Vector {
  Object elementData[] guarded_by this;
  int elementCount     guarded_by this;

  int lastIndexOf(Object o) {
    return lastIndexOf(o, elementCount - 1);
  }

  synchronized int lastIndexOf(Object o, int index) {
     …
  }

  …

}
```

RACE

IndexOutOfBoundsException

[TOPLAS 2006]

Controlling Thread Interference #2 Beyond Race Freedom

35

An Introduction to Programming with Threads

by Andrew D. Birrell

January 6, 1989

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

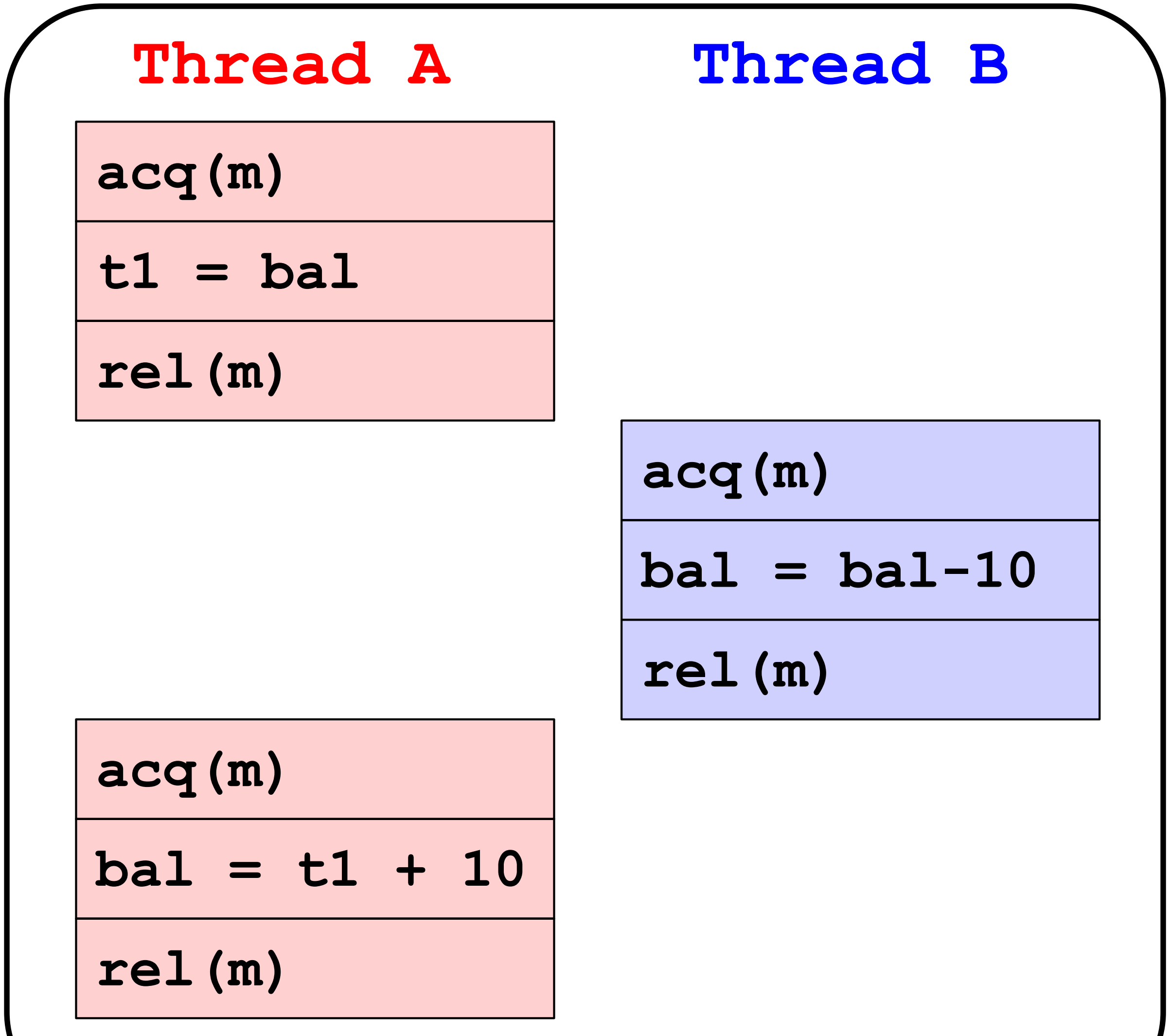
Race Freedom is not Enough

Thread A

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

acq(m);
bal = t1 + 10;
rel(m);
```

Thread B

```
...
acq(m);
bal = bal - 10;
rel(m);
```



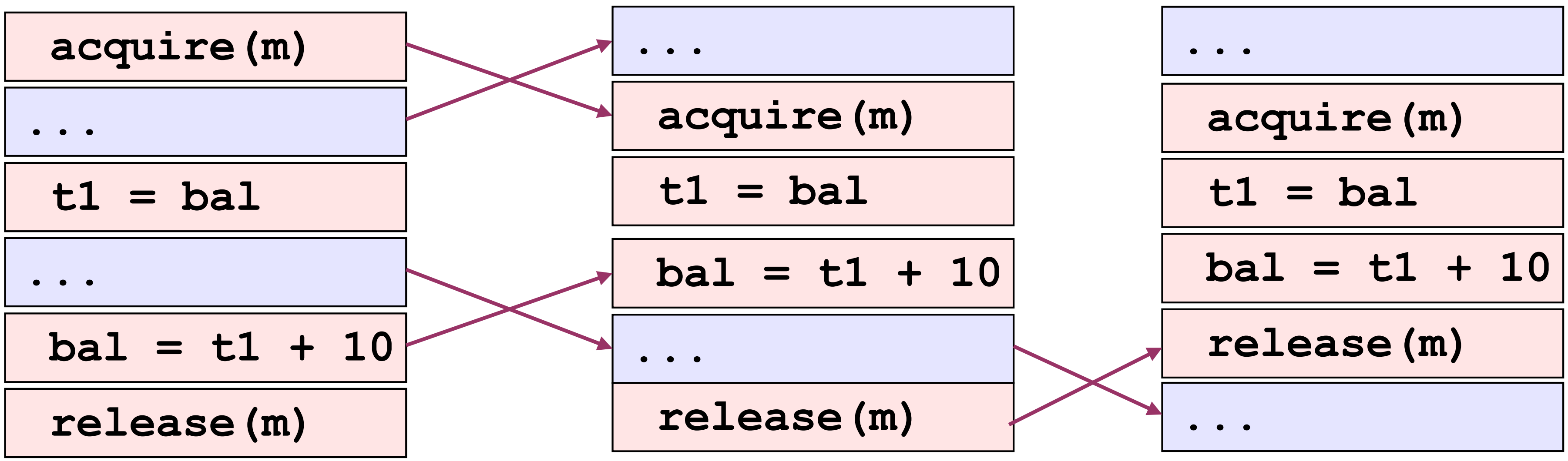
Controlling Thread Interference: #2 Enforce Atomicity

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

- sequential reasoning valid for atomic methods
- 90% of methods are atomic

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

Theory of Reduction [Lipton 76]



R	Right-mover	Acquire
L	Left-mover	Release
R+L	Both-mover	Race-Free Access
N	Non-mover	Racy Access

Serializable blocks have the pattern: R* [N] L*

A Type System for Atomicity

- Theorem 2
  - Any well-typed program behaves <u>*as if*</u> each atomic method executes serially (without interleaved steps of other threads) [toplas'08]

- Many other analyses for atomicity
  - FFY'08, FF'04, FFLQ'08, WS'06, XBH'06, PLZ'09, RDFHLR'05, FM'08

A Type System for Atomicity

- Many analyses for atomicity
  - FFY'08, FF'04, FFLQ'08, WS'06, XBH'06, PLZ'09, RDFHLR'05, FM'08

– Including a type system for atomicity
  - TOPLAS'08

- Theorem 2
  – Any well-typed program behaves *as if* each atomic method executes serially, without interleaved steps of other threads

java.lang.StringBuffer

```
/**
  ... used by the compiler to implement the binary
  string concatenation operator ...

  String buffers are safe for use by multiple
  threads. The methods are synchronized so that
  all the operations on any particular instance
  behave as if they occur in some serial order
  that is consistent with the order of the method
  calls made by each of the individual threads
  involved.
*/

public atomic class StringBuffer { ... }
```

java.lang.StringBuffer is not Atomic

```
public atomic StringBuffer {
  private int count guarded_by this;
  public synchronized int length() { return count; }
  public synchronized void getChars(...) { ... }

  public synchronized void append(StringBuffer sb){
R..L int len = sb.length();
       ...
       ...
R..L sb.getChars(...,len,...);
       ...
  }
}
```

sb.length() acquires the lock on sb, gets the length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

- violates pattern (R*[N]L*), append() is not atomic

Controlling Thread Interference
#3 Beyond Atomicity

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

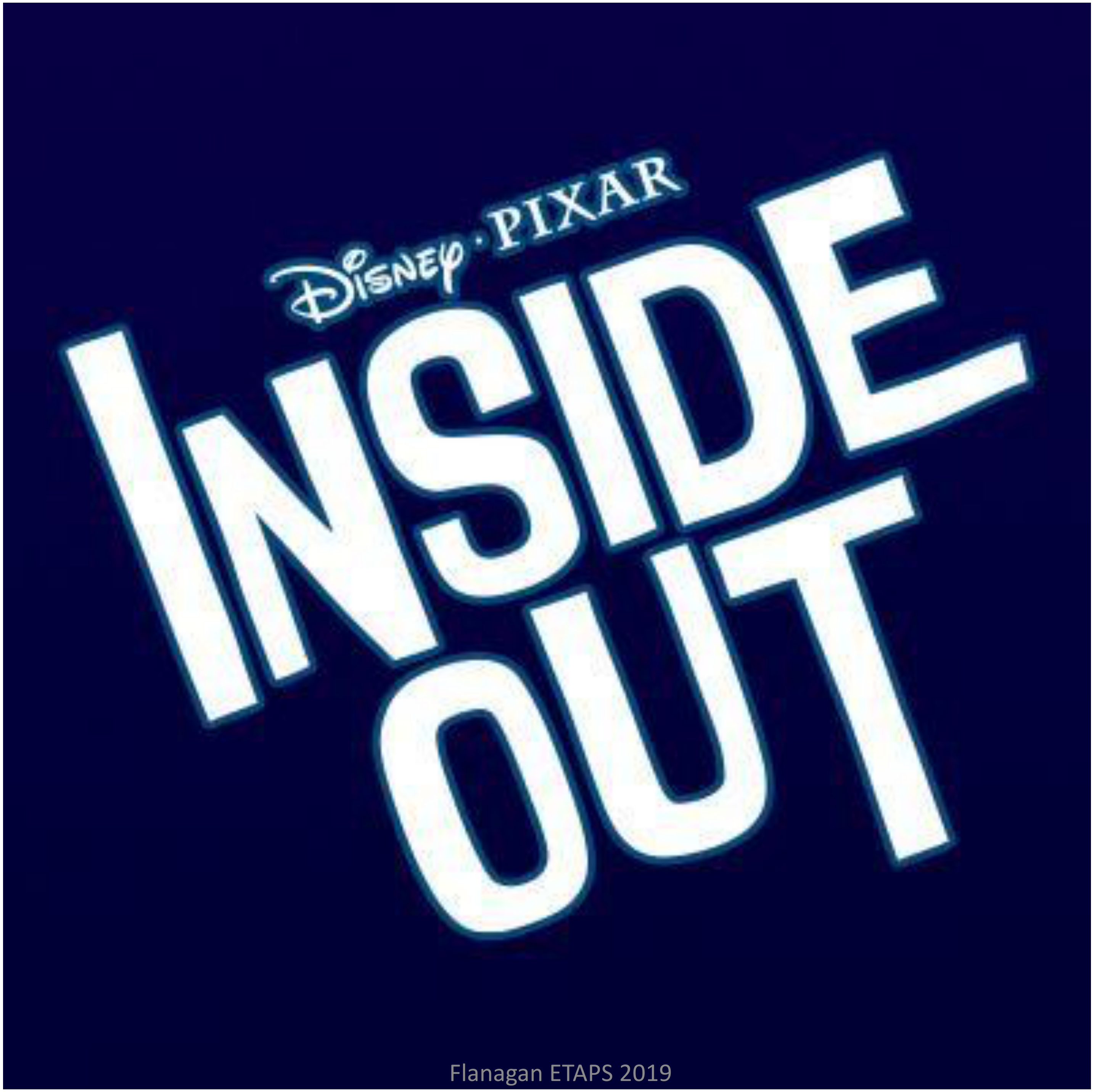
```
void busy_wait() {
  acq(m);
  thread interference?
  while (!test()) {
   thread interference?
   rel(m);
   thread interference?
   acq(m);
   thread interference?
   x++;
   thread interference?
  }
```

Two Semantics!

increment
vs.
non-atomic
read-modify-write

- ~90% of methods atomic
- Sequential reasoning

- ~10% of methods not atomic
- Pervasive interference
- Atomicity provides no help
- Local atomic blocks awakward


Disney·PIXAR
INSIDE
OUT

Controlling Thread Interference: #3 Explicit Yields

weird semantics

```
yield
```

```
atomic {
  ...
  ...
}
```

good semantics

```
...
...
```

weird semantics

```
yield
```

```
atomic {
  ...
  ...
}
```

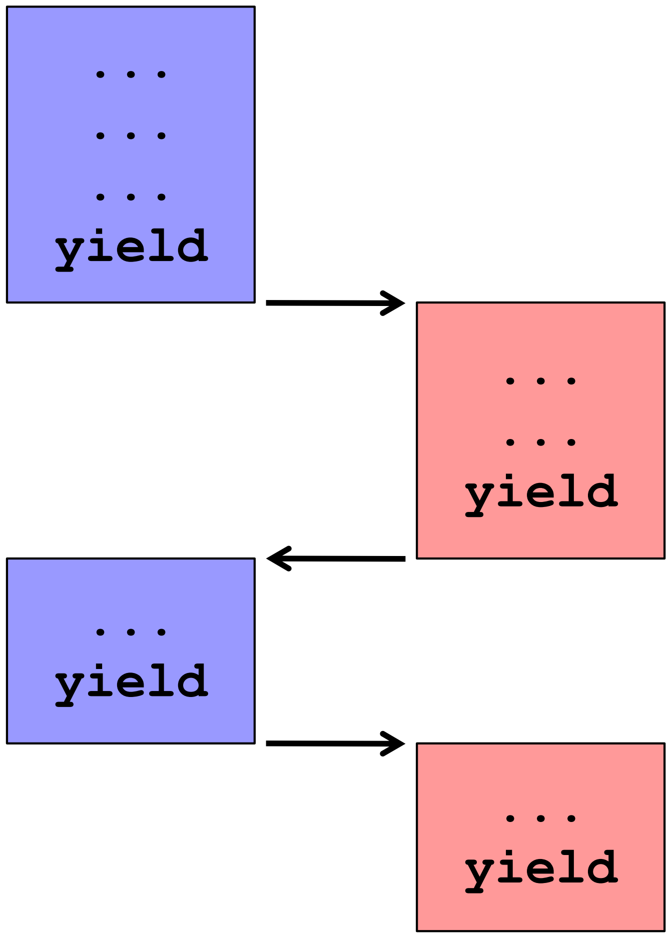
good semantics

```
...
...
```

```
yield
```

Non-Preemptive Scheduling

- Context switches only at yields
- Clean semantics
  - Sequential reasoning valid by default ...
  - ... except where yields highlight thread interference



- Limitation: Uses only a single processor

Non-Preemptive Scheduler

- Sequential reasoning
- Except where yields indicate interference

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

```
...
barrier
yield
```

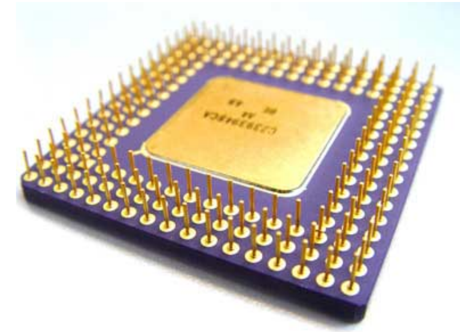
```
...
yield
```

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

Code with explicit yields

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

Preemptive/ non-preemptive equivalence verified by analyses



Preemptive Scheduler

- Full performance
- No overhead

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

```
...
barrier
yield
```

```
...
yield
```

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

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++;
}
```

Explicit Yields

```
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++;

}
```

A Type System for Preemptive/non-preemptive equivalence

- Theorem 3
  - Any well-typed program behaves *as if* run on a non-preemptive scheduler (even when run on preemptive/multicore hardware)

- Other analyses
  - eg IB'07, YF'10, YSF'11, CCHRRRT'17

```
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;
  }

  ...
}
```

- Yields help programmers identify defects
  - difference is statistically significant
  - [Sadowski, Yi PLATEAU 2010]

Review of Non-interference Specs

- Race freedom
  - code behaves **as if** on sequentially consistent memory model
- Atomicity
  - code behaves **as if** atomic methods executed serially (~90%)
- Yield-oriented programming
  - code behaves **as if** run on non-preemptive scheduler
  - sequential reasoning ok ...
  - ... except where yields indicate thread interference (1-10/KLOC)
  - http://users.soe.ucsc.edu/~cormac/coop.html

Analysis Tools for Non-Interference

Analysis Tools for Non-Interference

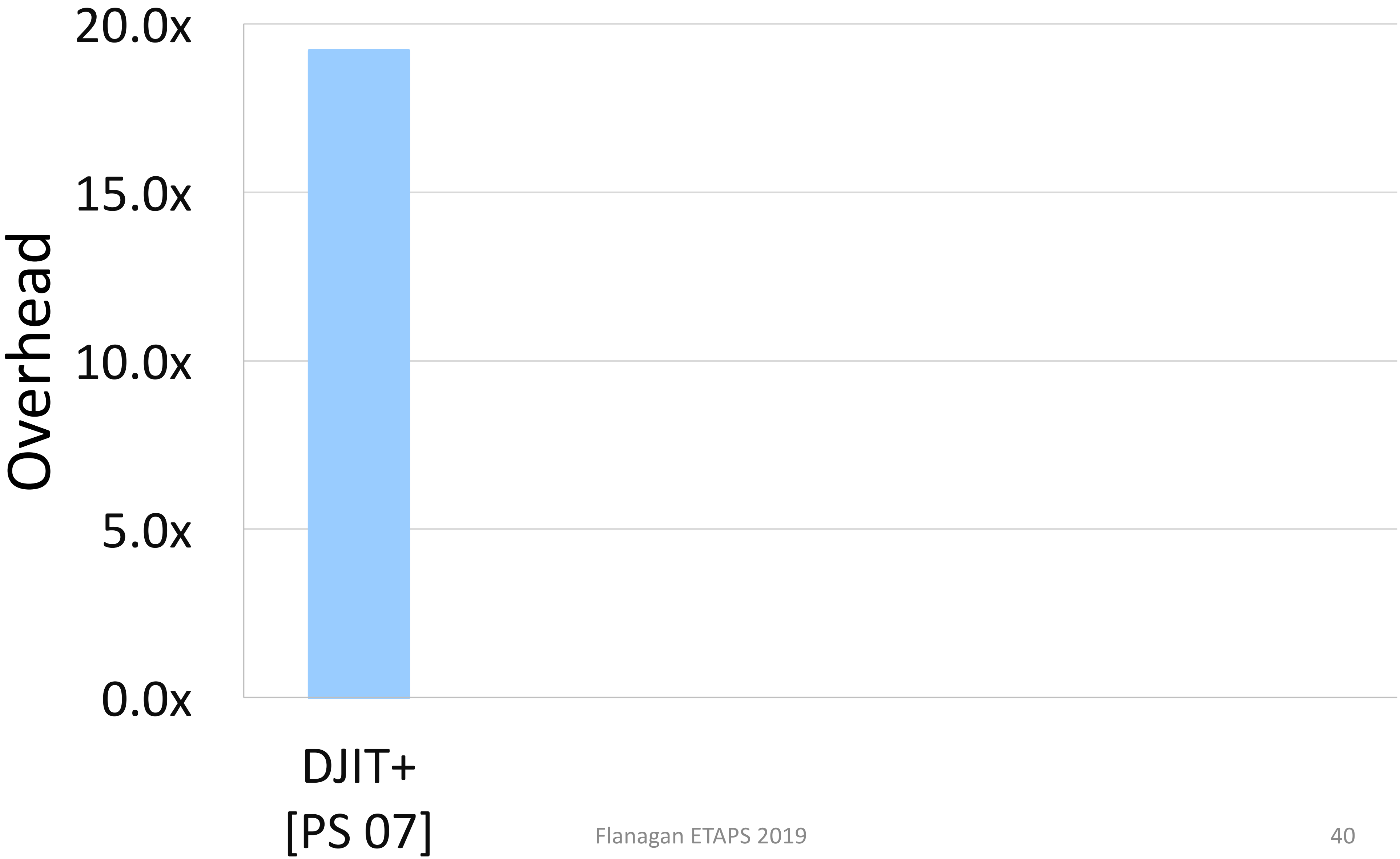
Static analysis

- observe syntax
- over-approximate behavior
- report all errors (theorems!)
- report (many?) false alarms

Dynamic analysis

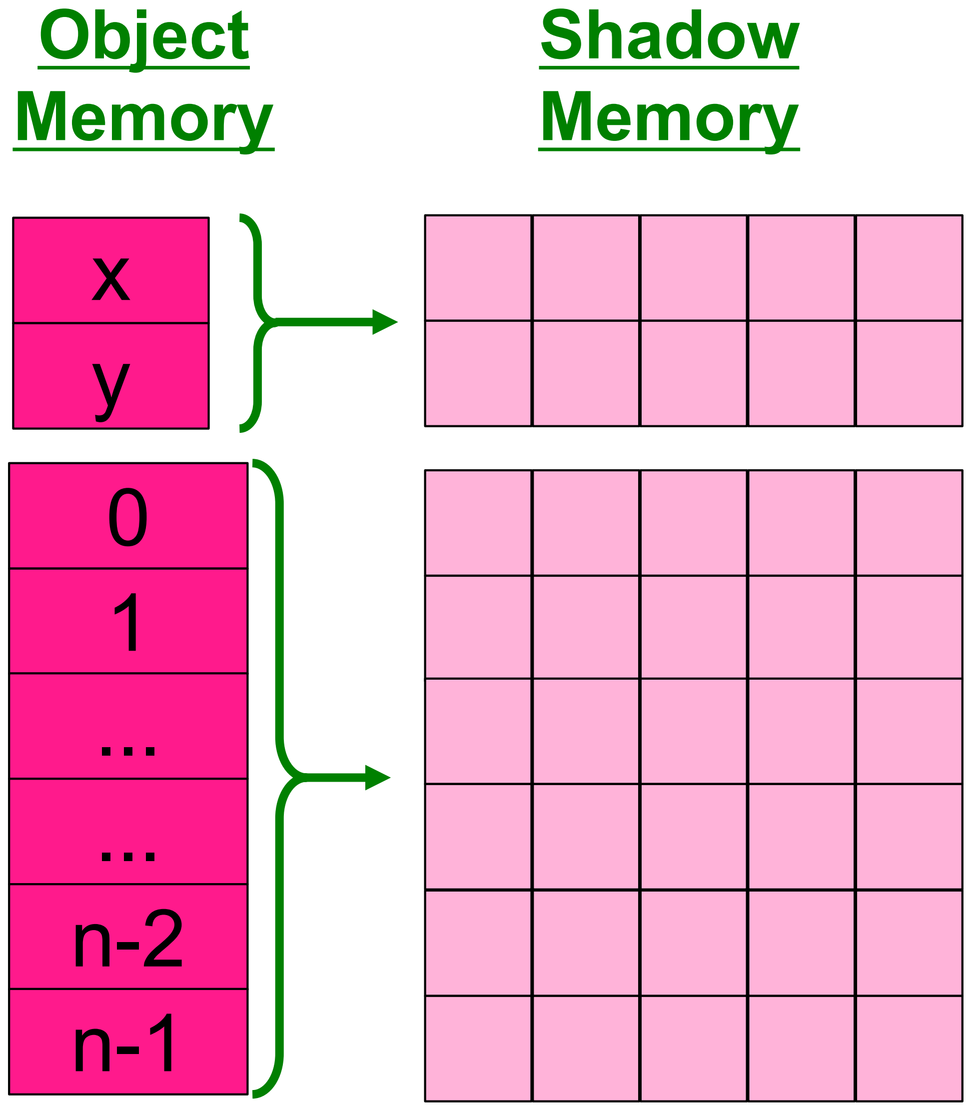
- observe traces
- under-approximate behavior
- miss some errors
- can guarantee no false alarms

Precise Dynamic Race Detection



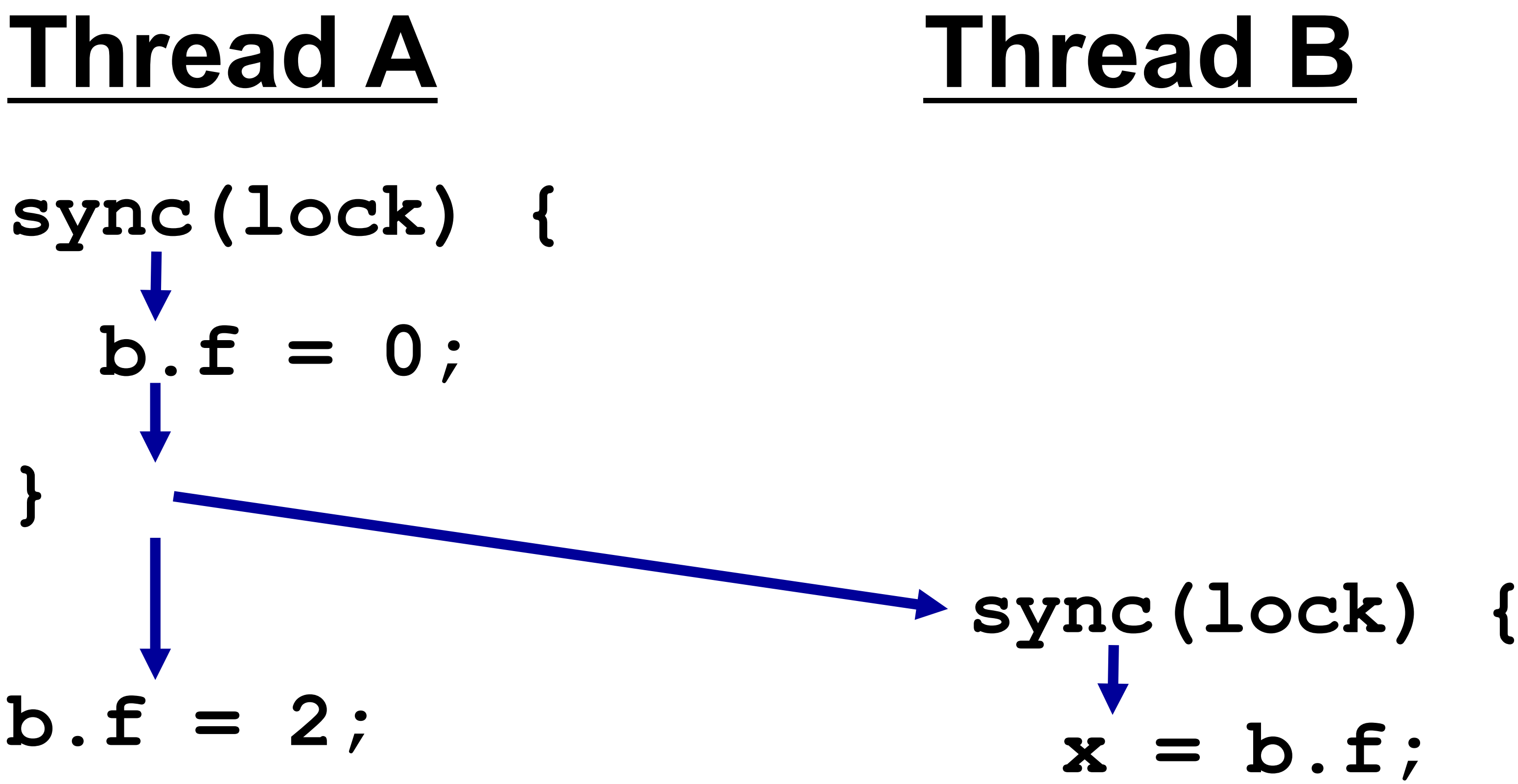
Dynamic Race Detection Overhead

```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    check(this.x); tmp = this.x;
    check(this.x); this.x = tmp + dx;
    check(this.y); tmp = this.y;
    check(this.y); this.y = tmp + dy;
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      check(a[i]); a[i]=0;
    }
  }
}
```



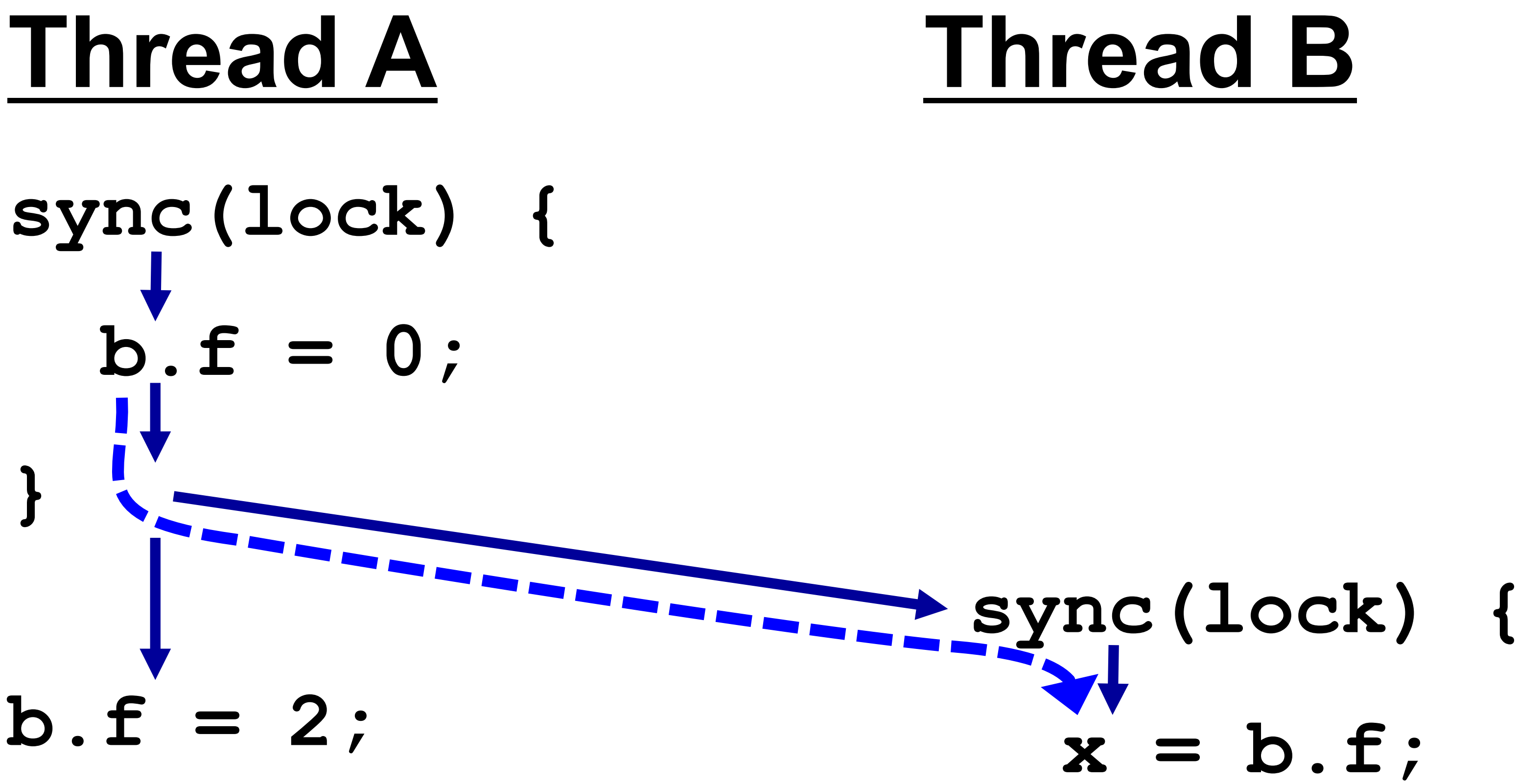
Data Races

- Happens-Before Relation [Lamport 78]



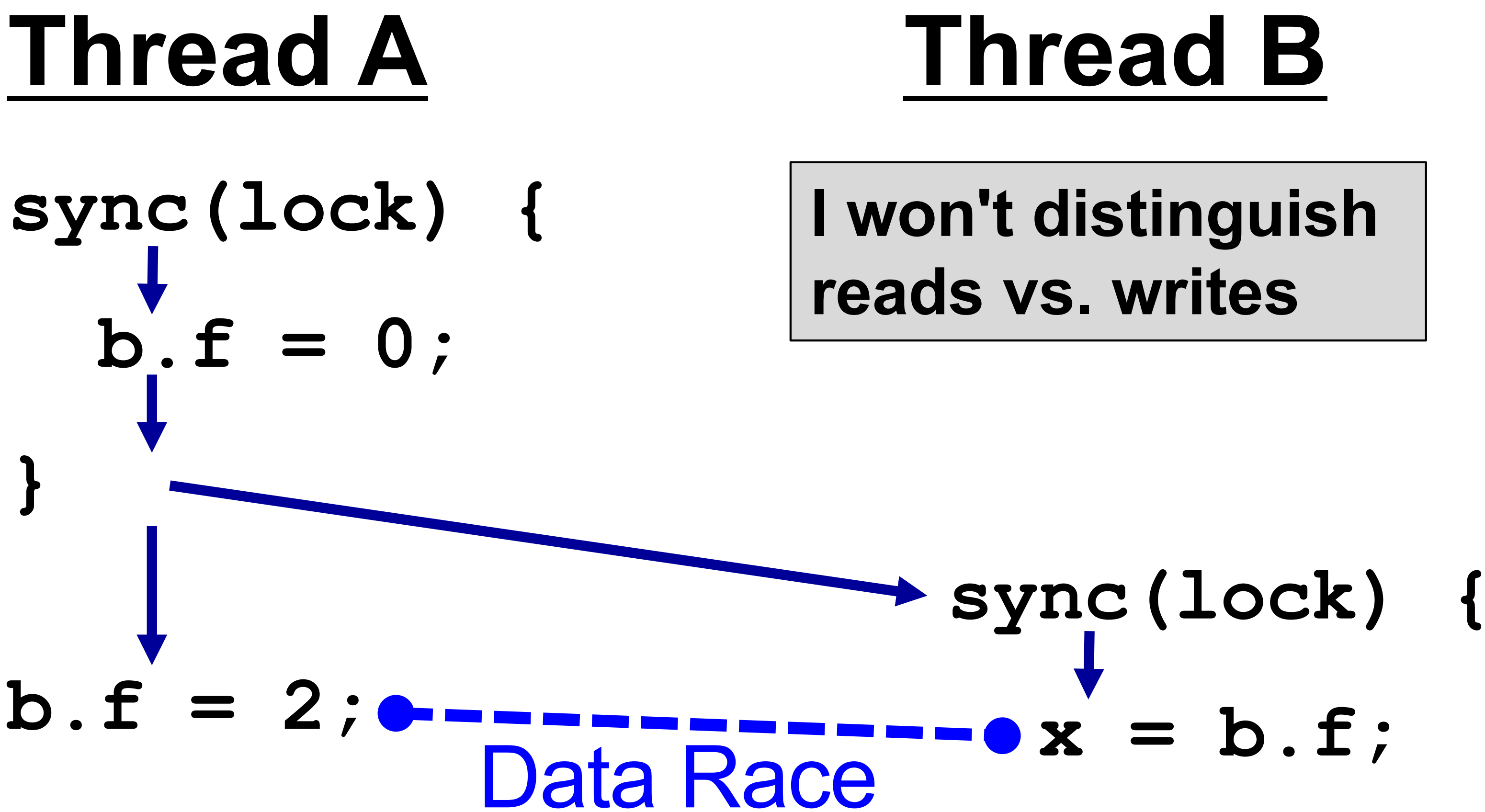
Data Races

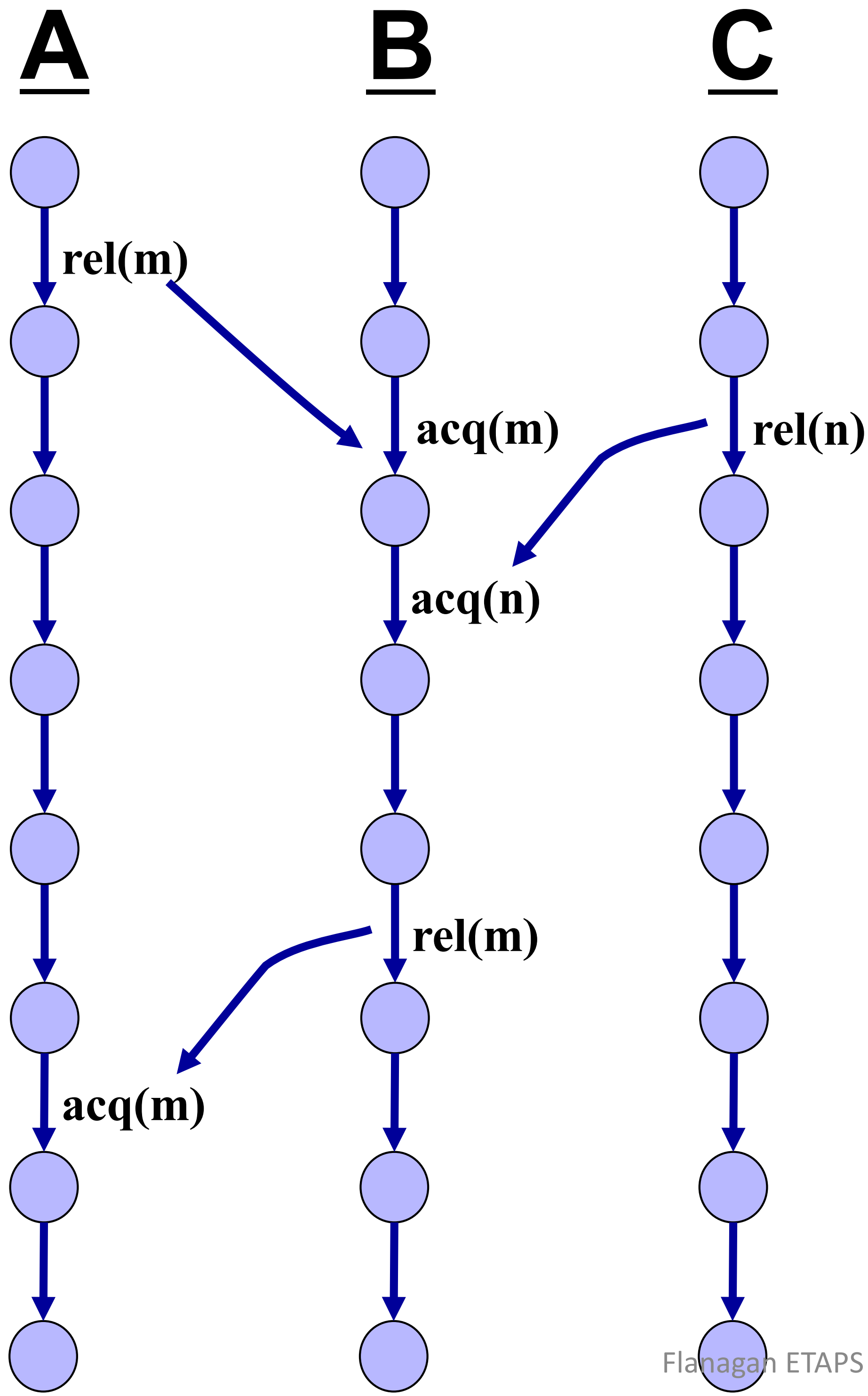
- Happens-Before Relation [Lamport 78]
- Data Race: unordered accesses



Data Races

- Happens-Before Relation [Lamport 78]
- Data Race: unordered accesses

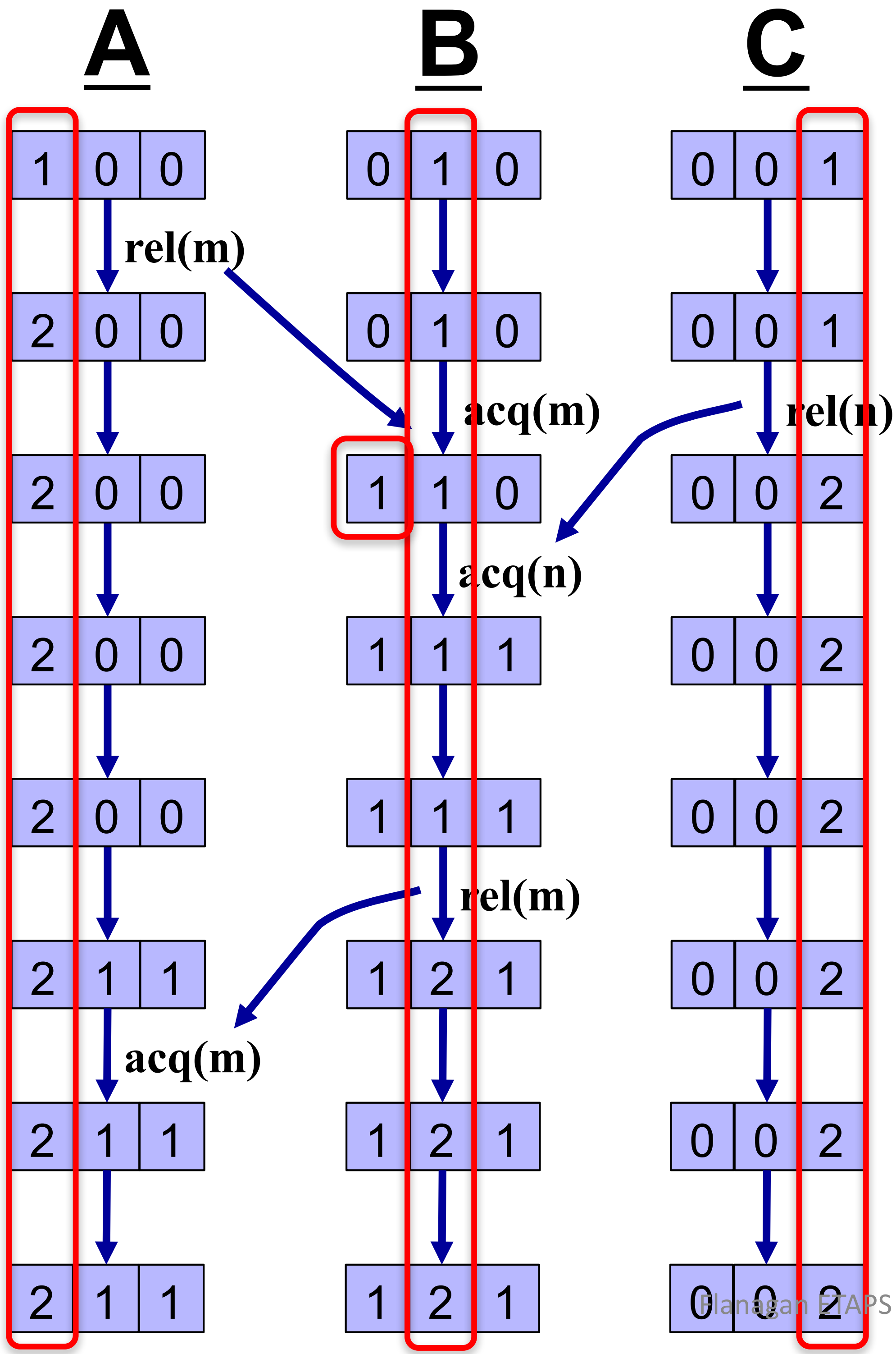


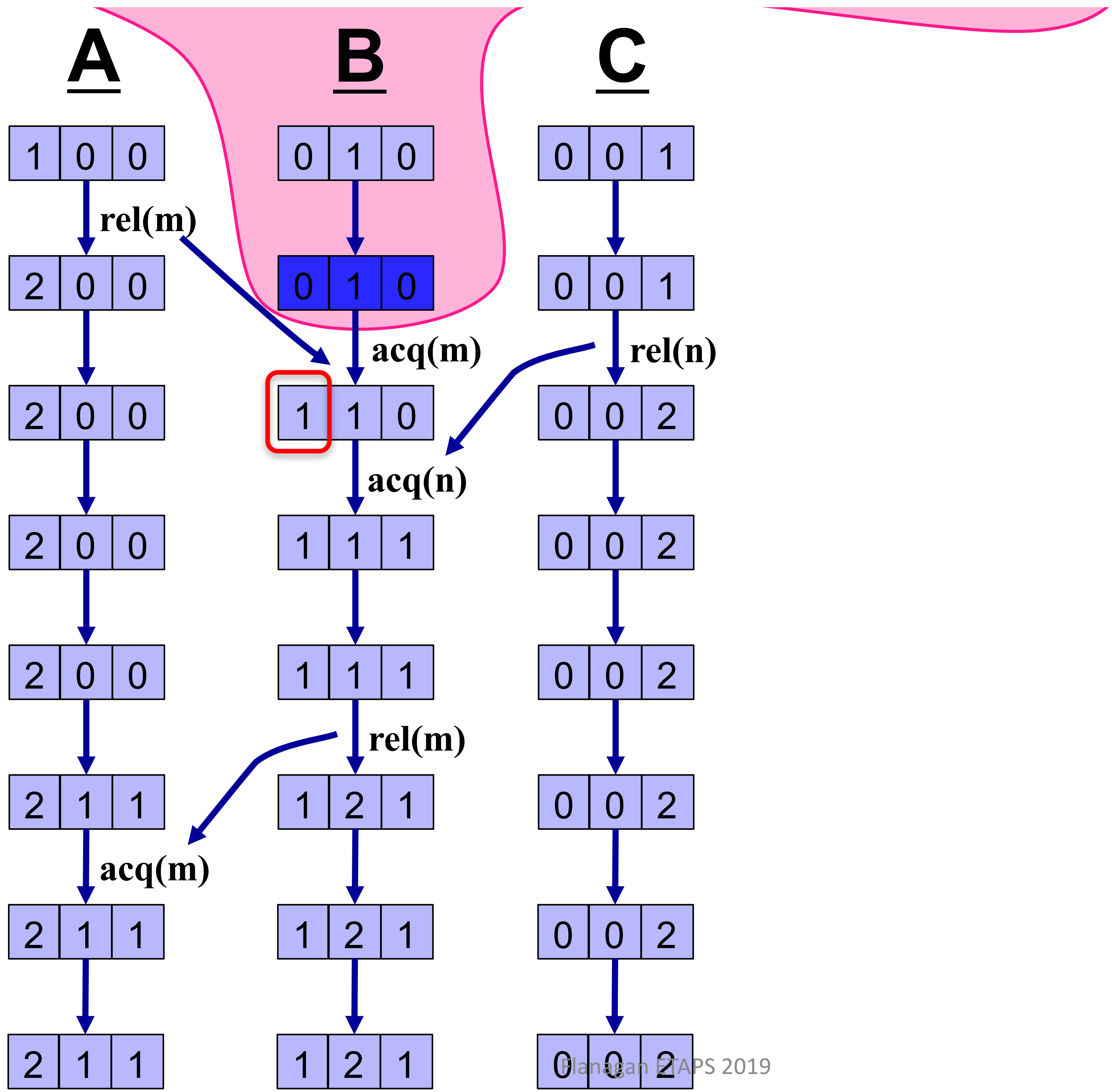


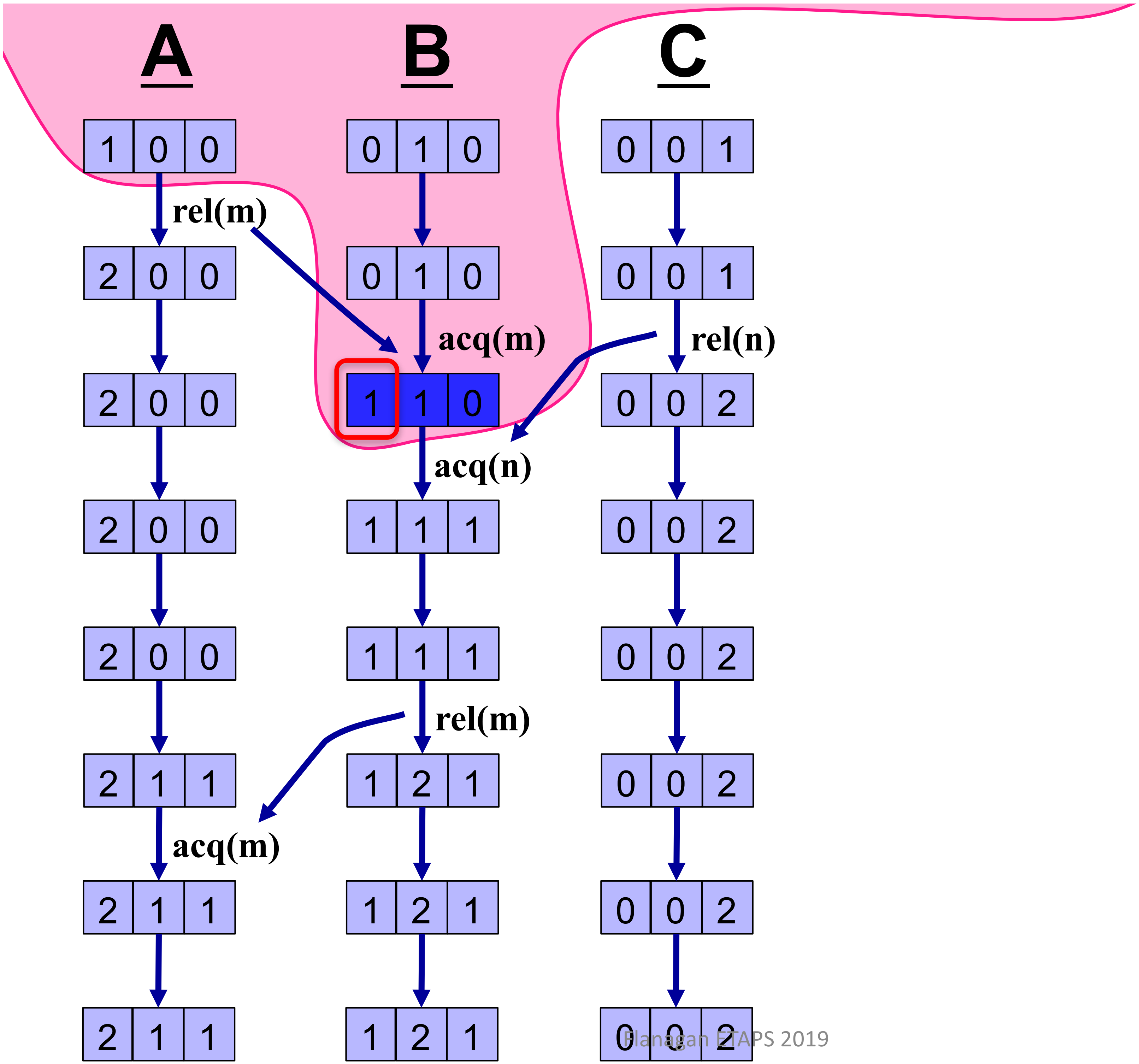
Tracking the Happens-Before Relation

- Program Order
- Synchronization Order

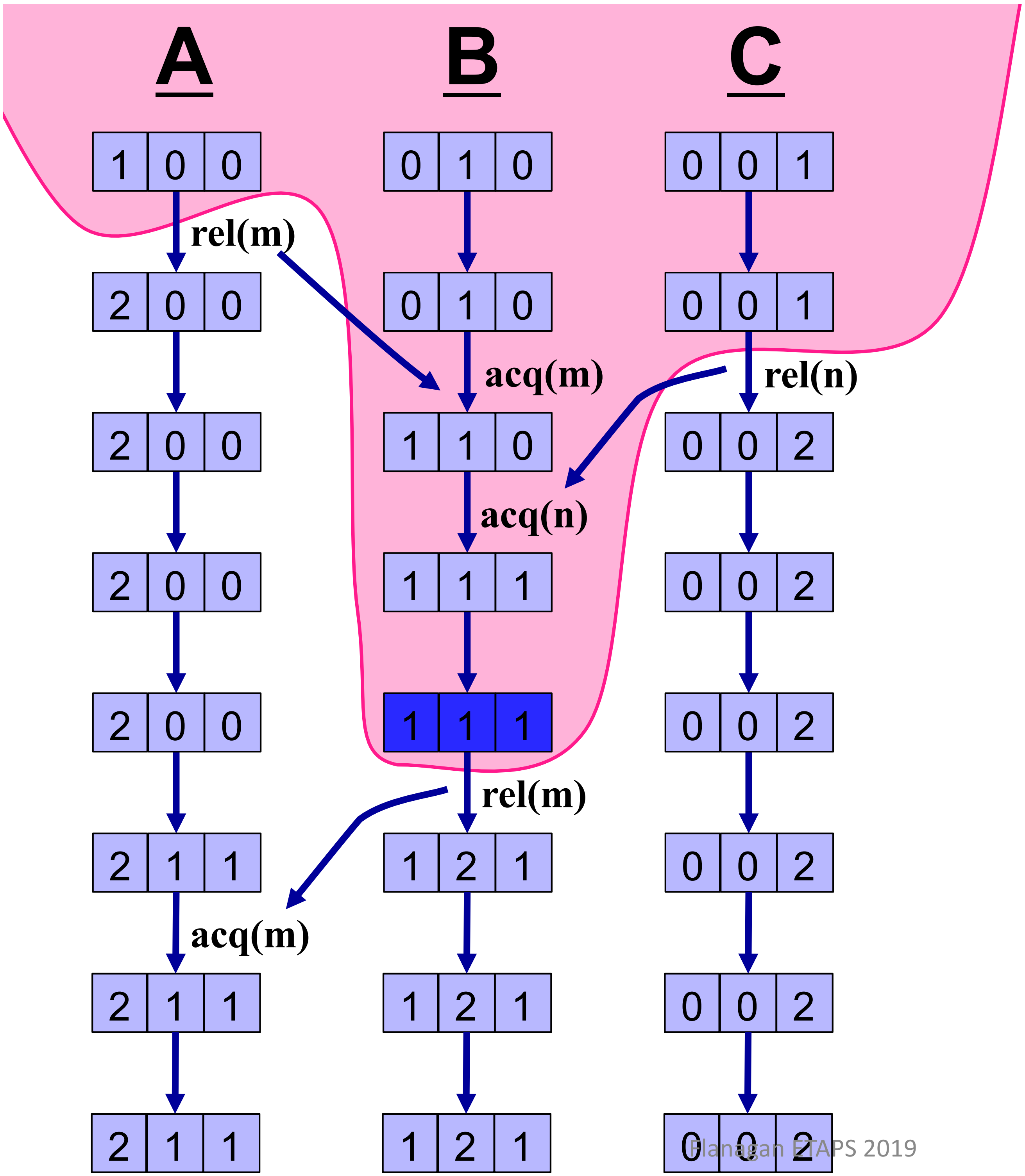
Vector Clocks [Mattern 88]

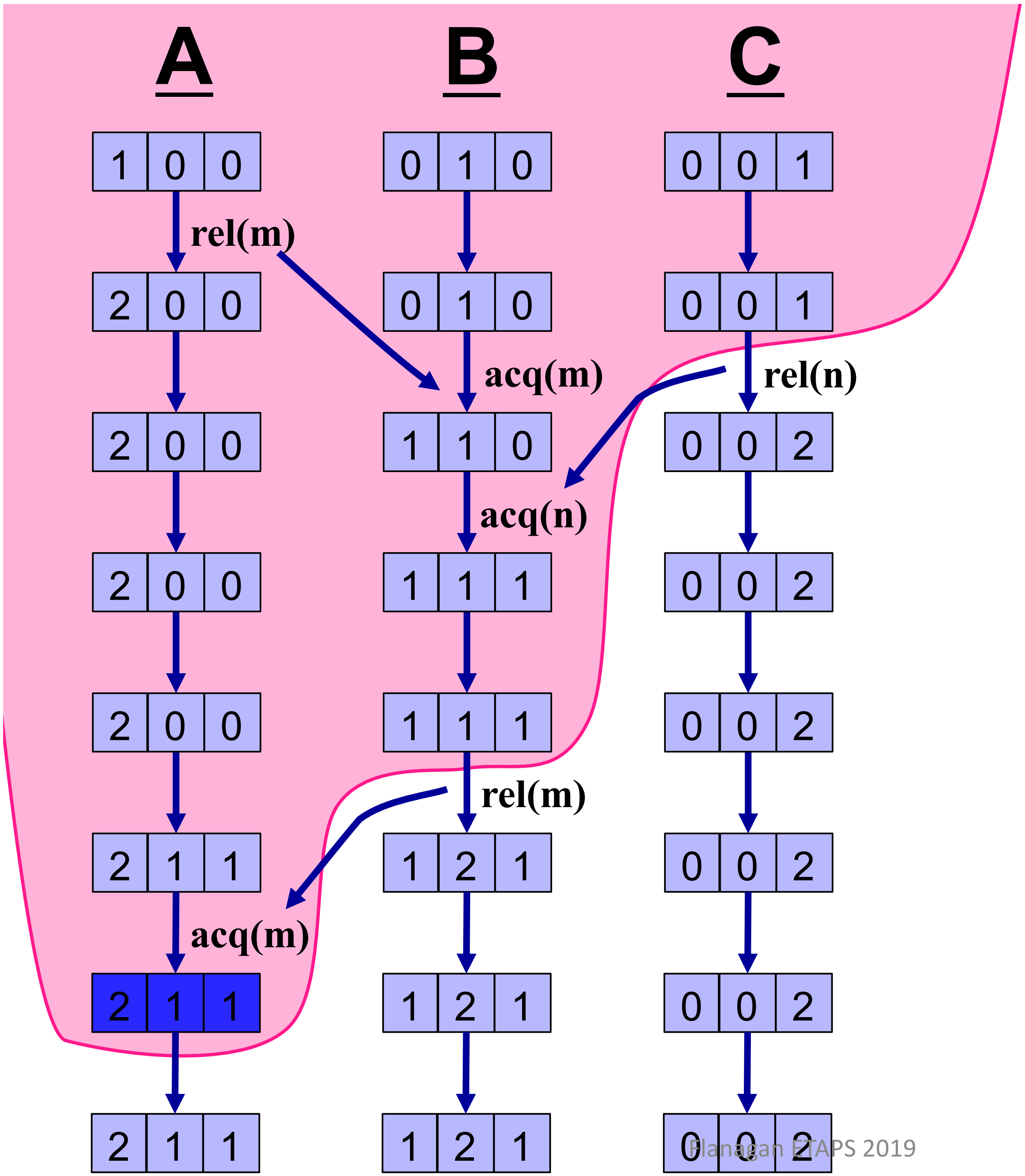


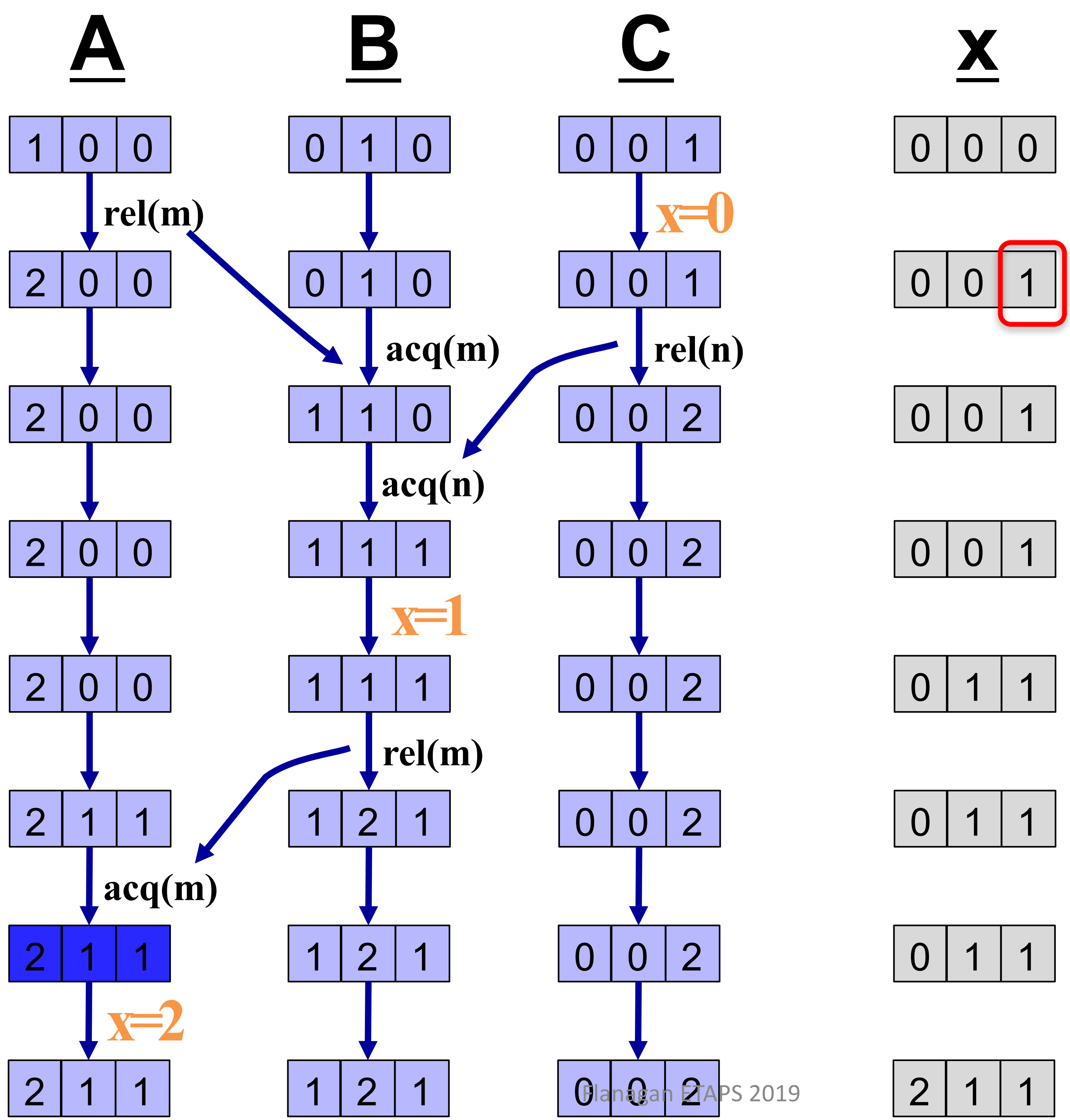

A
B
C
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 1 0
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
acq(n)
2 0 0
1 1 1
0 0 2
2 0 0
1 1 1
0 0 2
rel(m)
2 1 1
1 2 1
0 0 2
acq(m)
2 1 1
1 2 1
0 0 2
2 1 1
1 2 1
0 0 2



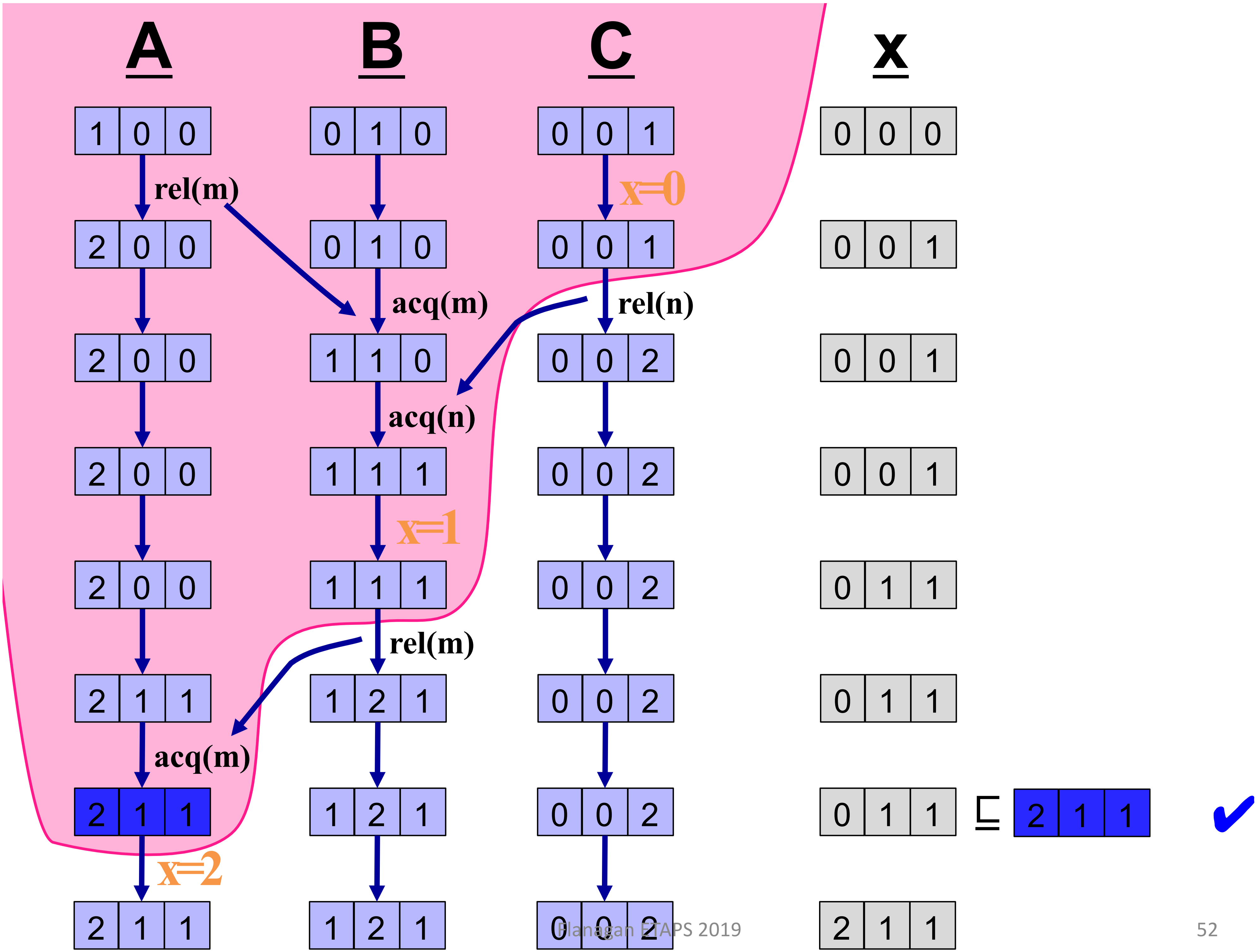
A
B
C
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 1 0
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
acq(n)
2 0 0
1 1 1
0 0 2
2 0 0
1 1 1
0 0 2
rel(m)
2 1 1
1 2 1
0 0 2
acq(m)
2 1 1
1 2 1
0 0 2
2 1 1
1 2 1
0 0 2


A
B
C
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 1 0
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
acq(n)
2 0 0
1 1 1
0 0 2
2 0 0
1 1 1
0 0 2
rel(m)
2 1 1
1 2 1
0 0 2
acq(m)
2 1 1
1 2 1
0 0 2
2 1 1
1 2 1
0 0 2

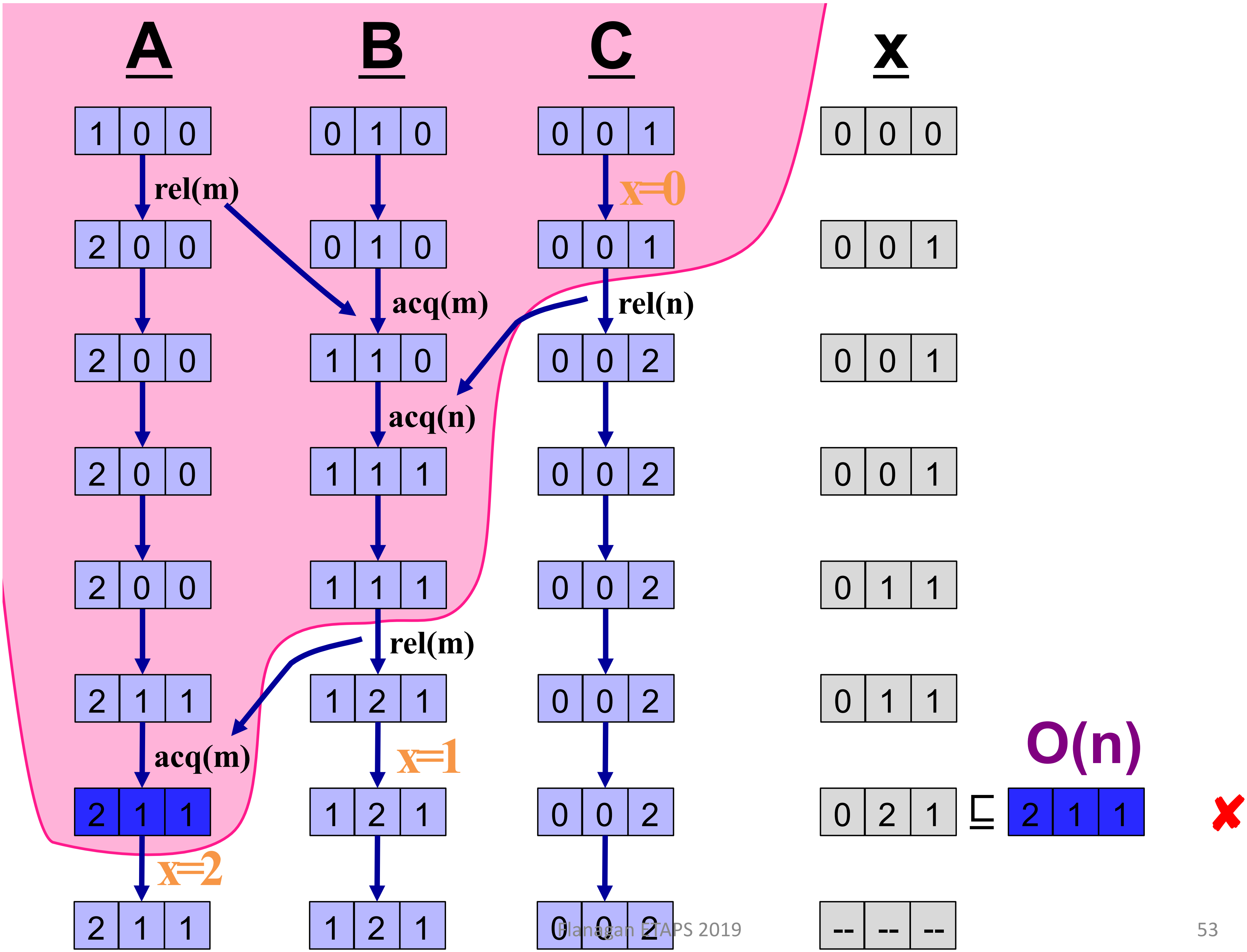

A
B
C
1 0 0
0 1 0
0 0 1
rel(m)
2 0 0
0 1 0
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
acq(n)
2 0 0
1 1 1
0 0 2
2 0 0
1 1 1
0 0 2
rel(m)
2 1 1
1 2 1
0 0 2
acq(m)
2 1 1
1 2 1
0 0 2
2 1 1
1 2 1
0 0 2



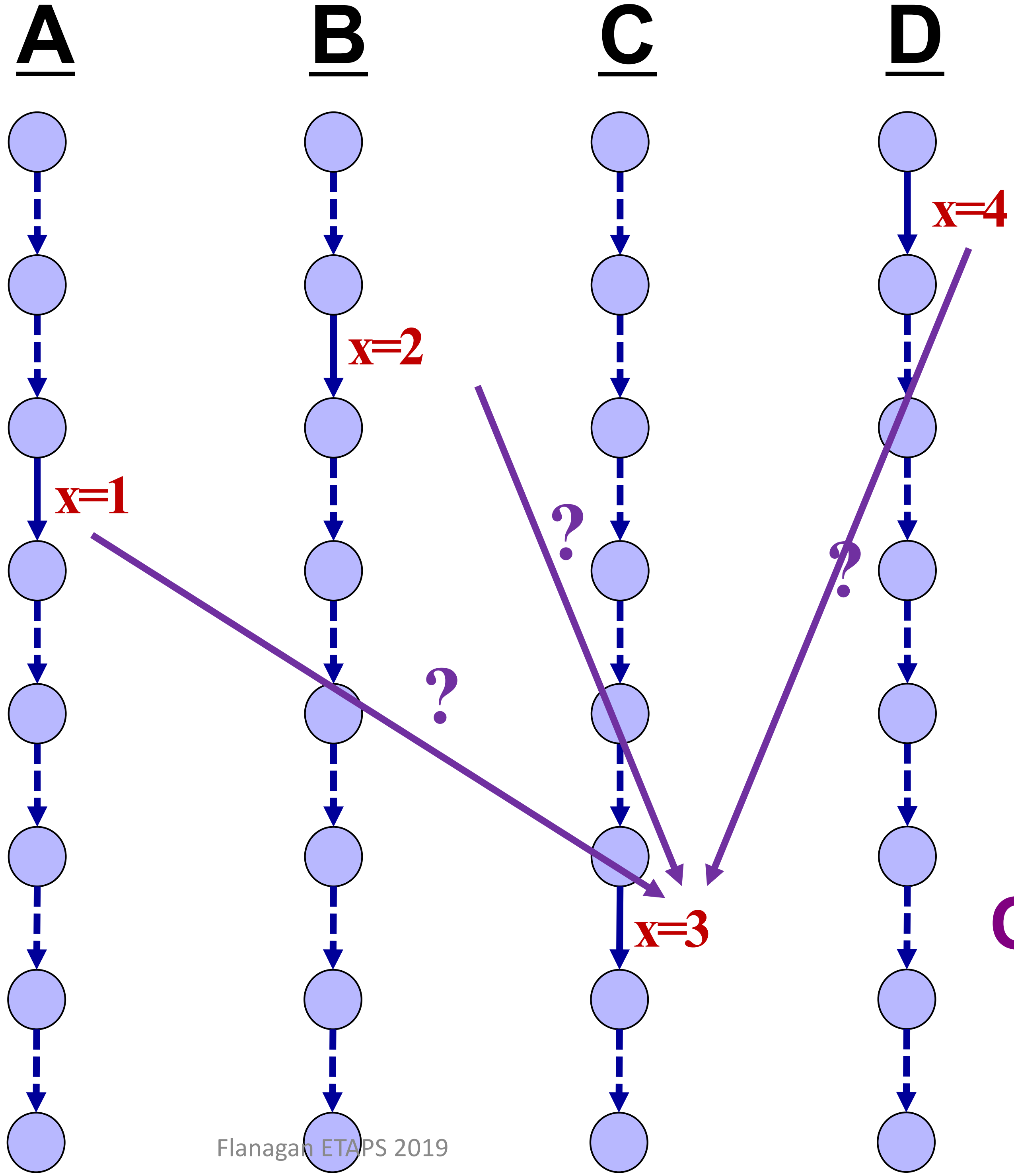
A
B
C
x
1 0 0
0 1 0
0 0 1
0 0 0
rel(m)
x=0
2 0 0
0 1 0
0 0 1
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
0 0 1
acq(n)
2 0 0
1 1 1
0 0 2
0 0 1
x=1
2 0 0
1 1 1
0 0 2
0 1 1
rel(m)
2 1 1
1 2 1
0 0 2
0 1 1
acq(m)
2 1 1
1 2 1
0 0 2
0 1 1
x=2
2 1 1
1 2 1
0 0 2
2 1 1



A
B
C
x
1 0 0
0 1 0
0 0 1
0 0 0
rel(m)
x=0
2 0 0
0 1 0
0 0 1
0 0 1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
0 0 1
acq(n)
2 0 0
1 1 1
0 0 2
0 0 1
x=1
2 0 0
1 1 1
0 0 2
0 1 1
rel(m)
2 1 1
1 2 1
0 0 2
0 1 1
acq(m)
2 1 1
1 2 1
0 0 2
0 1 1 ⊑ 2 1 1
✔
x=2
2 1 1
1 2 1
0 0 2
2 1 1

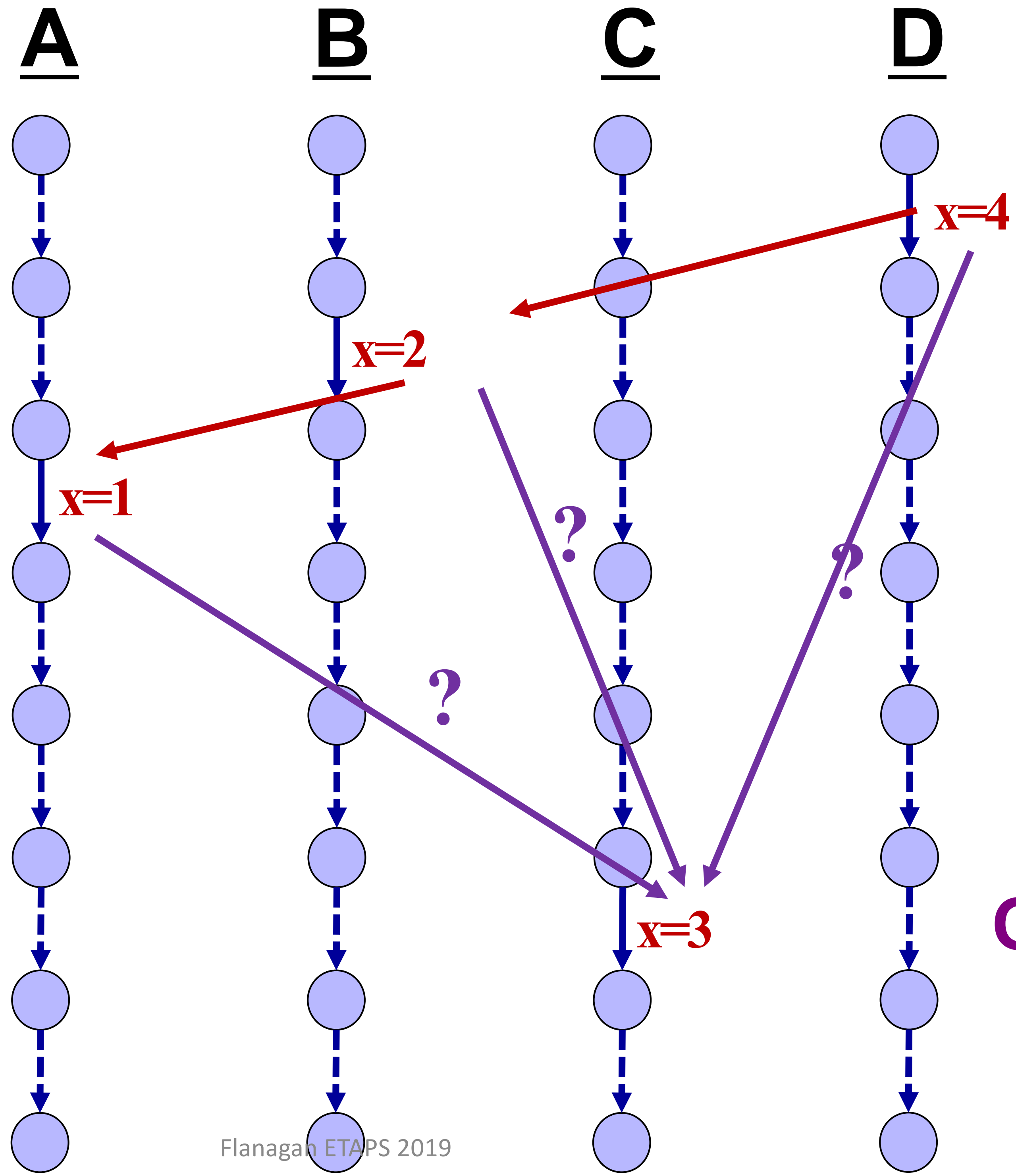

A
B
C
x
rel(m)
acq(m)
x=0
rel(n)
acq(n)
rel(m)
acq(m)
x=1
x=2
O(n)

Vector Clock Checks



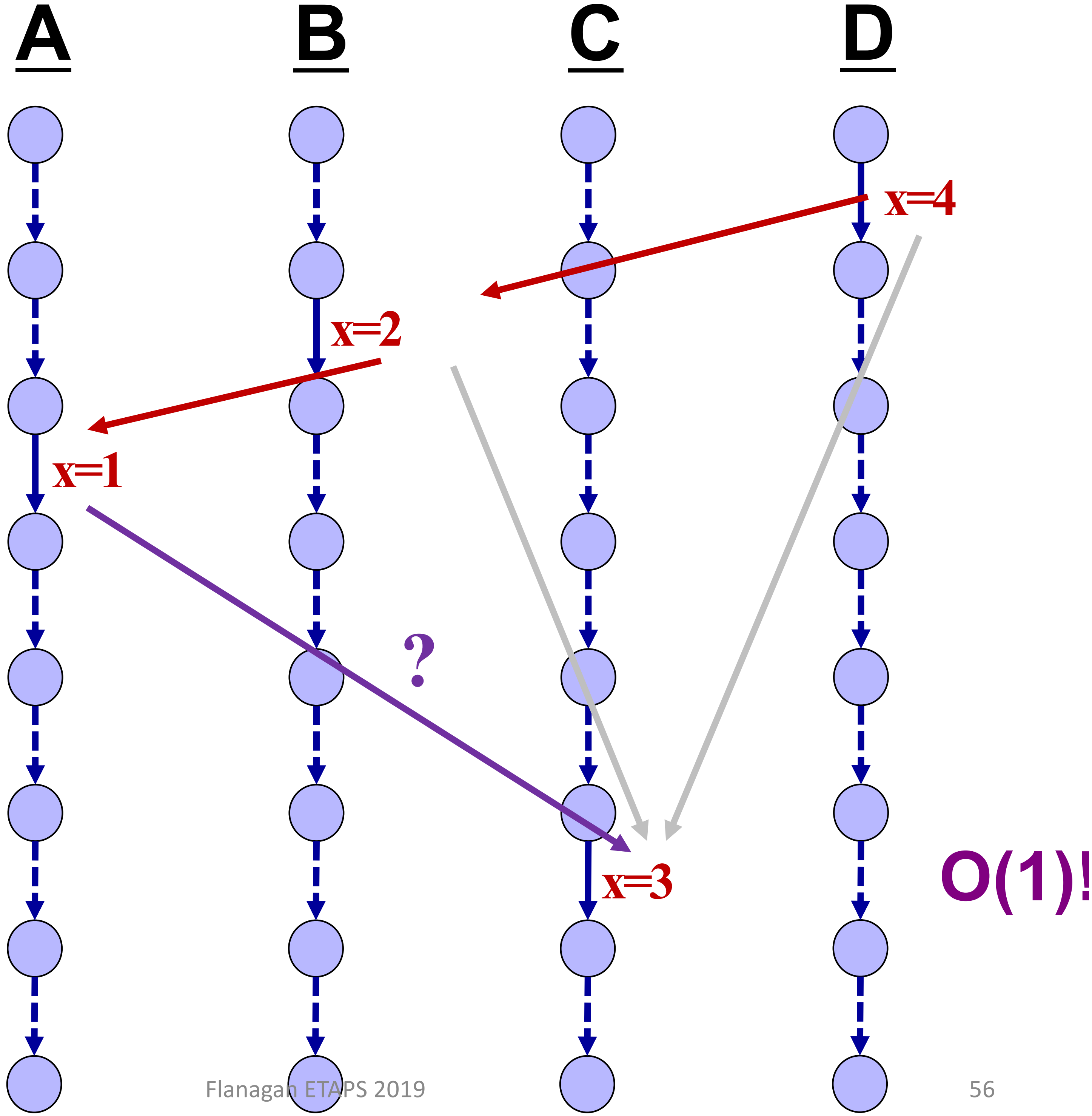
O(n)

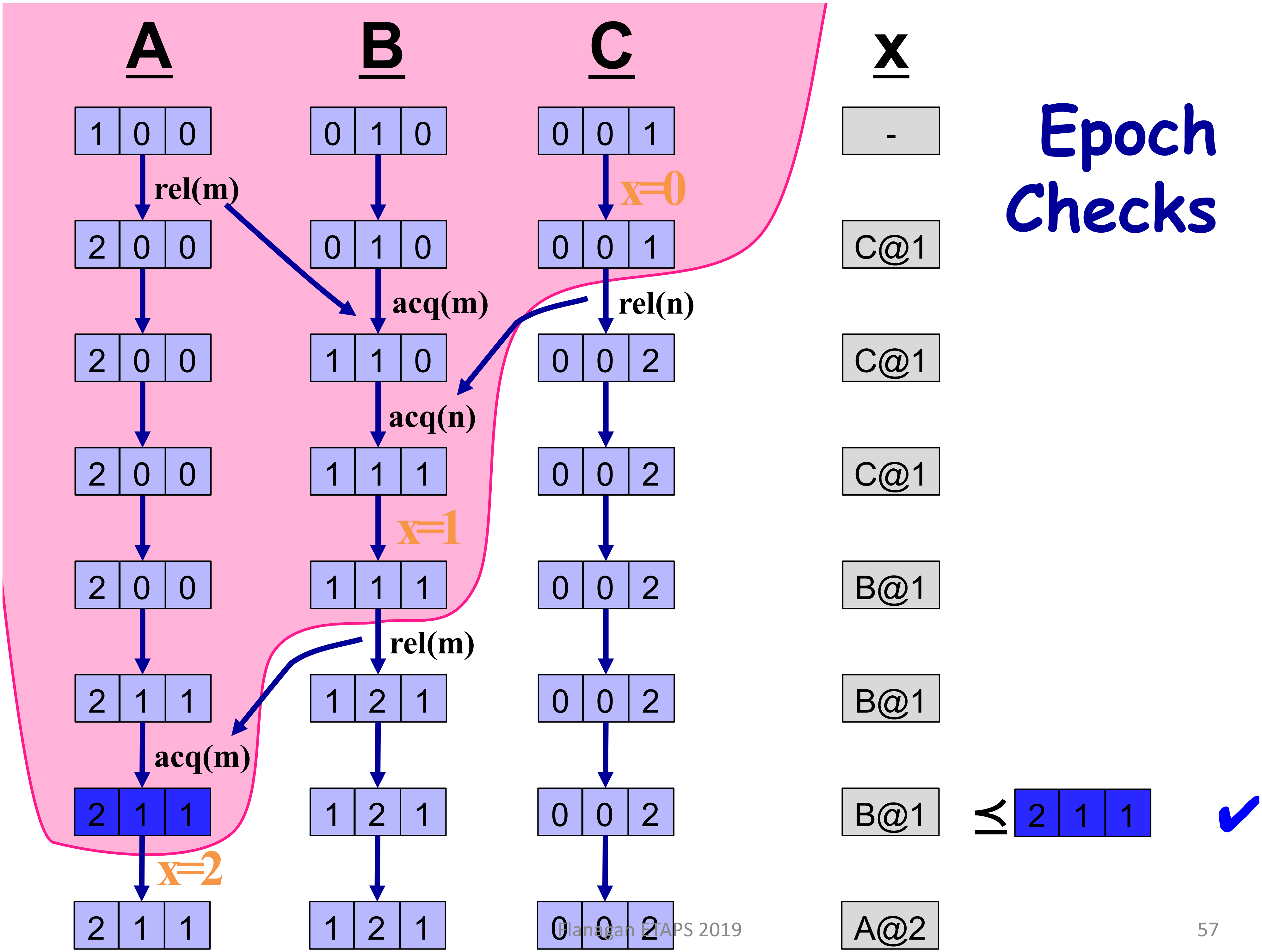
Vector Clock Checks



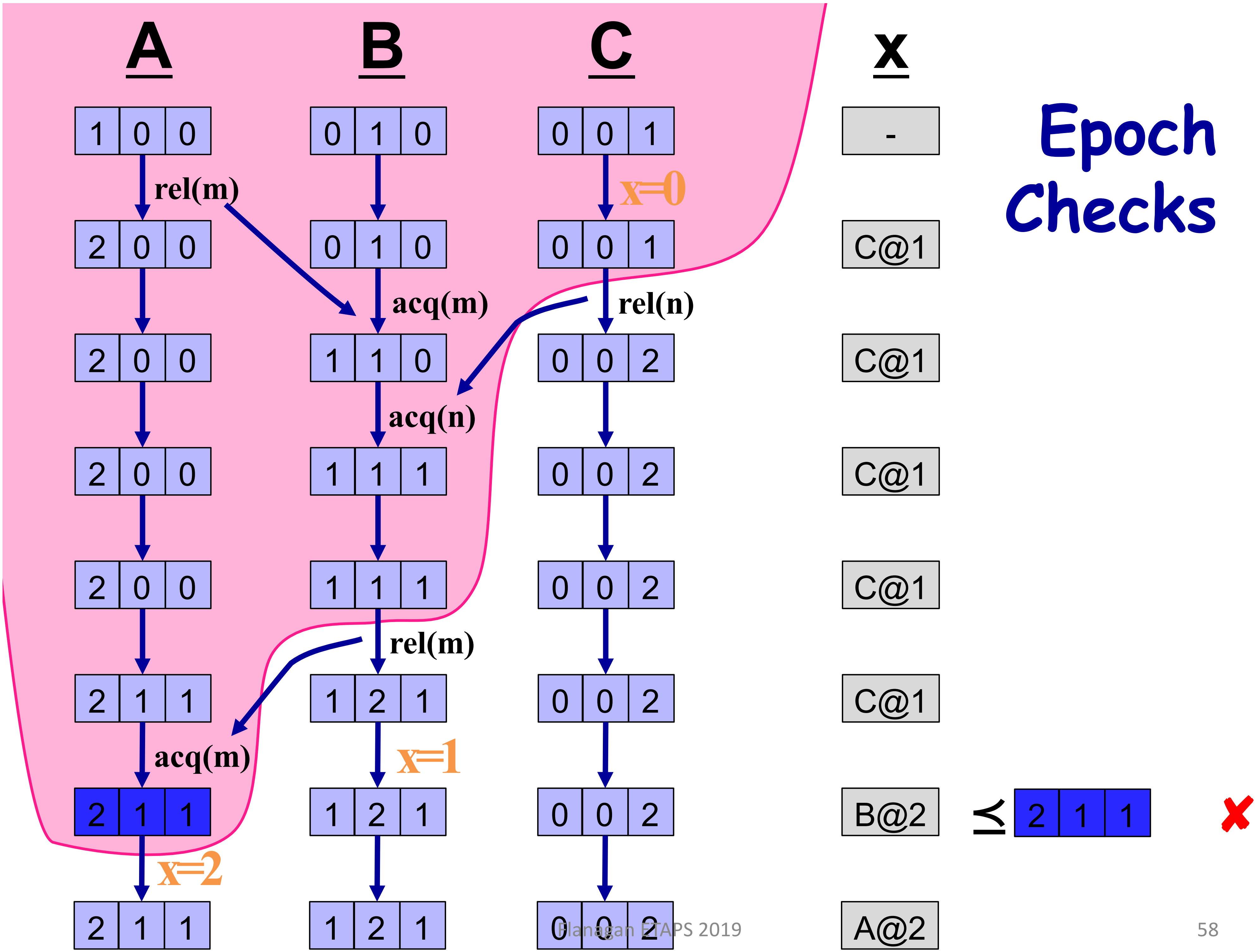
O(n)

Vector Clock Checks



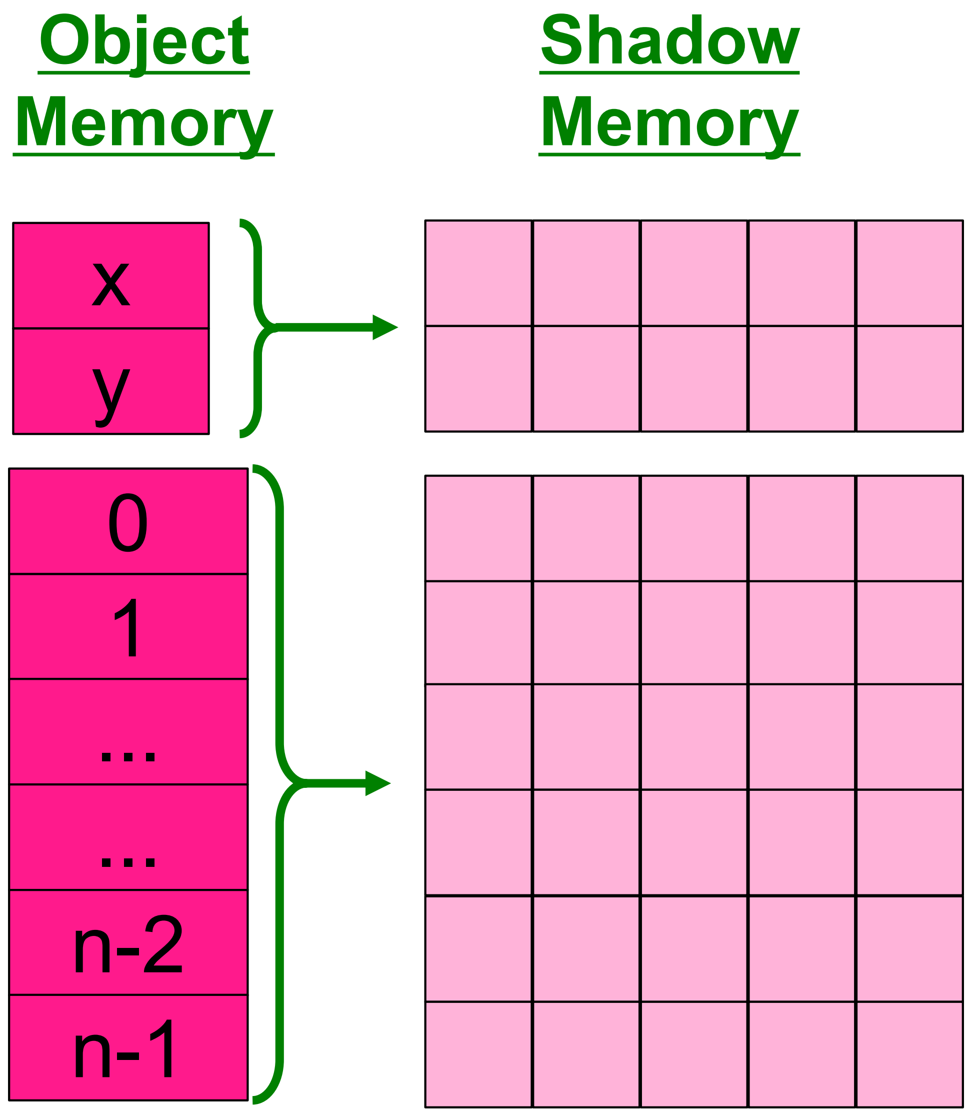

A
B
C
x
1 0 0
0 1 0
0 0 1
-
rel(m)
x=0
2 0 0
0 1 0
0 0 1
C@1
acq(m)
rel(n)
2 0 0
1 1 0
0 0 2
C@1
acq(n)
2 0 0
1 1 1
0 0 2
C@1
x=1
2 0 0
1 1 1
0 0 2
B@1
rel(m)
2 1 1
1 2 1
0 0 2
B@1
acq(m)
2 1 1
1 2 1
0 0 2
B@1
≼ 2 1 1
✔
x=2
2 1 1
1 2 1
0 0 2
A@2
Epoch Checks

Epoch Checks



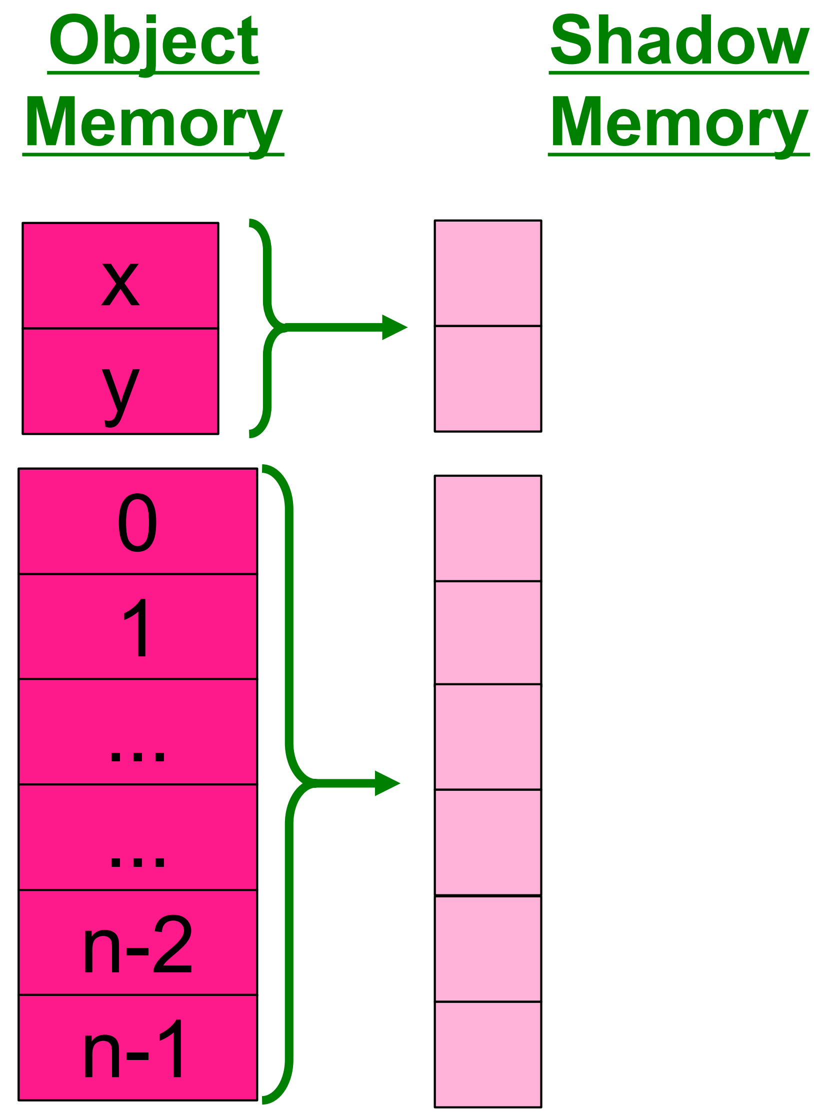
Dynamic Race Detection Overhead

```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    check(this.x); tmp = this.x;
    check(this.x); this.x = tmp + dx;
    check(this.y); tmp = this.y;
    check(this.y); this.y = tmp + dy;
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      check(a[i]); a[i]=0;
    }
  }
}
```

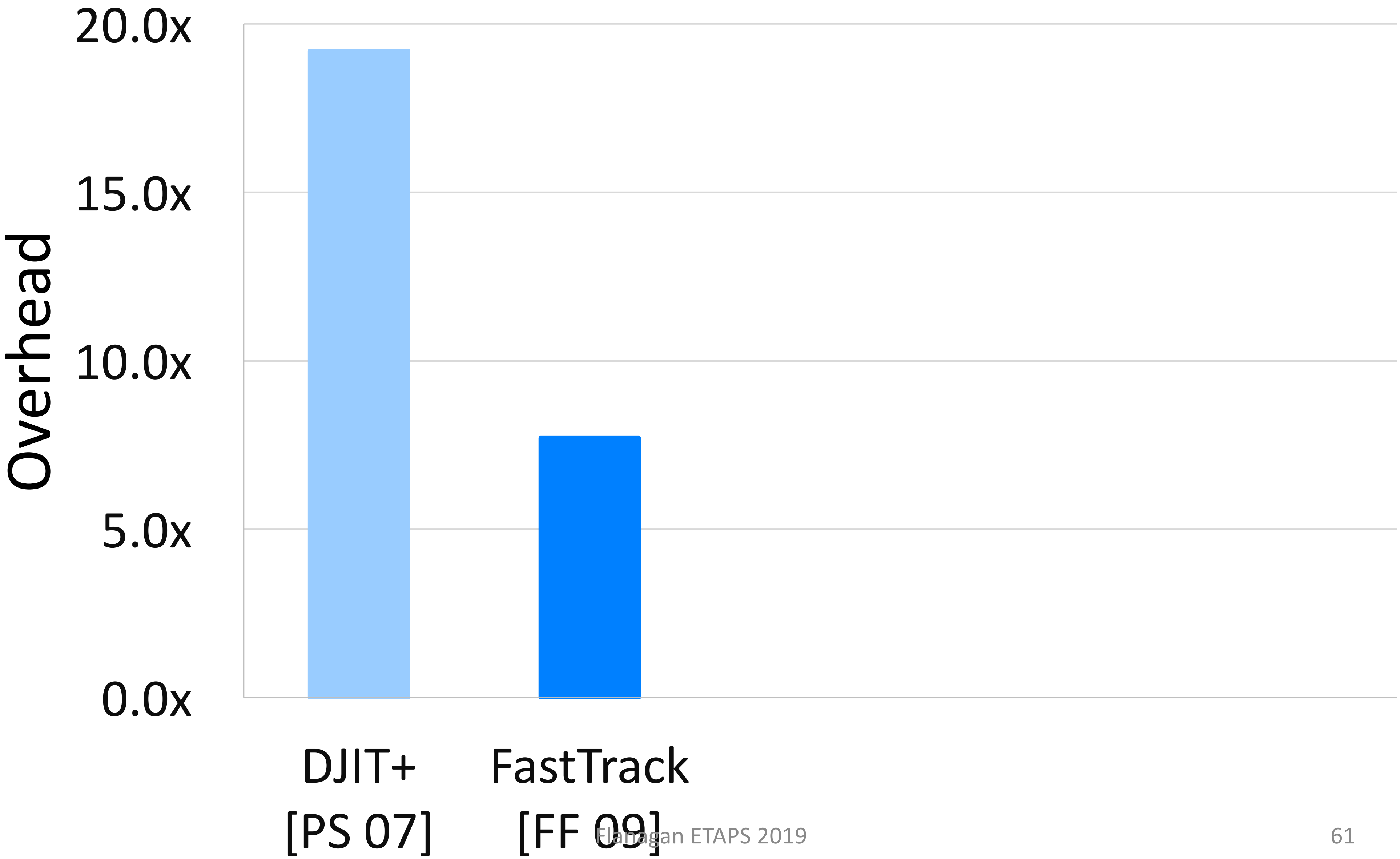


Dynamic Race Detection Overhead

```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    check(this.x); tmp = this.x;
    check(this.x); this.x = tmp + dx;
    check(this.y); tmp = this.y;
    check(this.y); this.y = tmp + dy;
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      check(a[i]); a[i]=0;
    }
  }
}
```

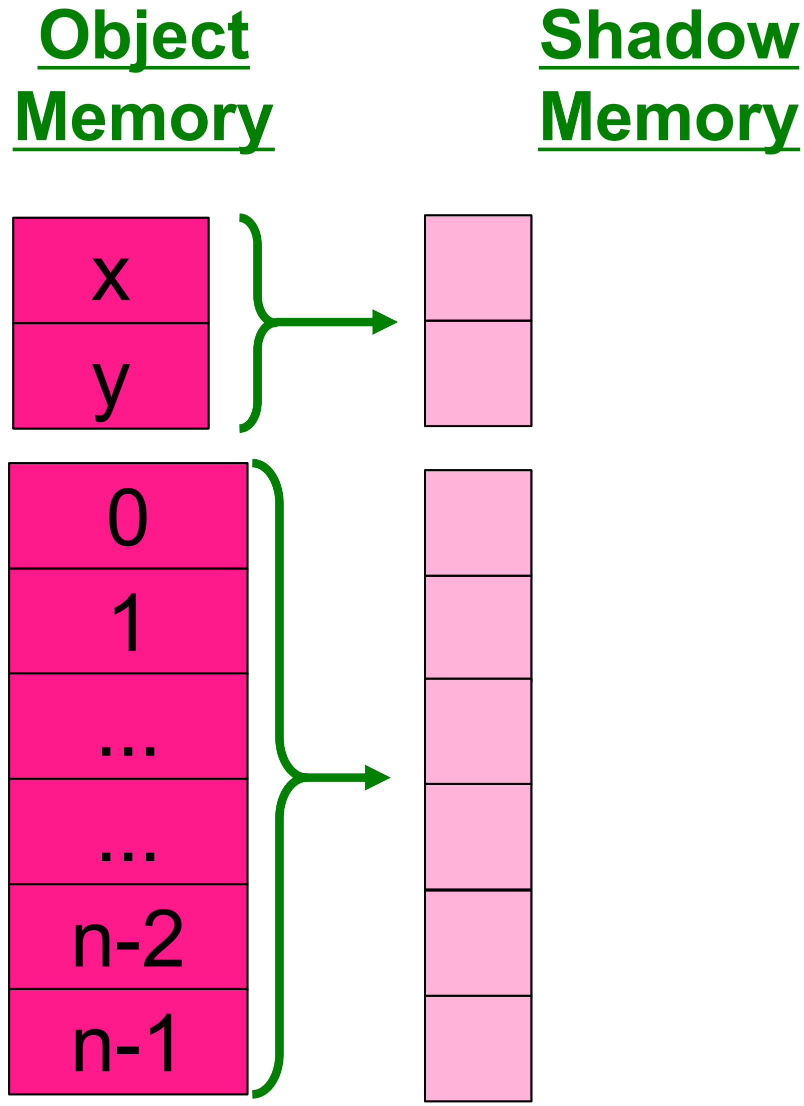


Precise Dynamic Race Detection



FastTrack Check Placement: 7.3x

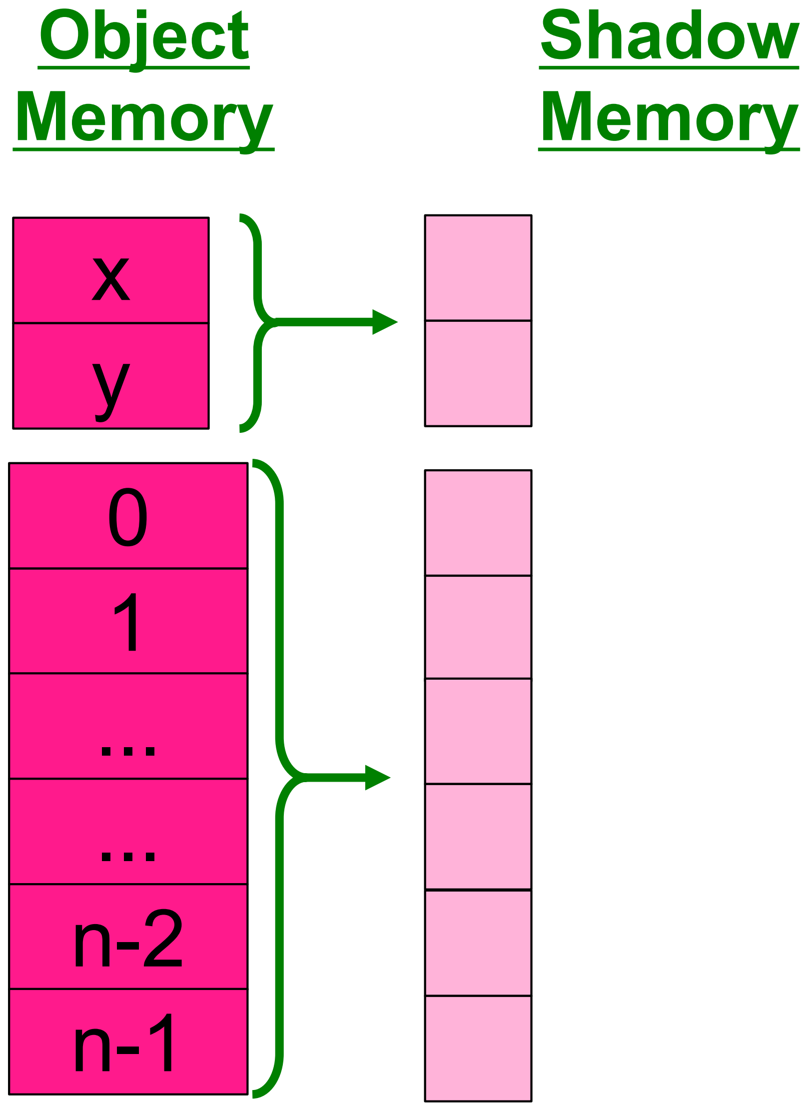
```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    check(this.x); tmp = this.x;
    check(this.x); this.x = tmp + dx;
    check(this.y); tmp = this.y;
    check(this.y); this.y = tmp + dy;
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      check(a[i]); a[i]=0;
    }
  }
}
```



BigFoot Check Placement: 2.5x

```
class Point {
 int x,y;
 void move(int dx, int dy) {
  int tmp;
  check(this.x); tmp = this.x;
  check(this.x); this.x = tmp + dx;
  check(this.y); tmp = this.y;
  check(this.y); this.y = tmp + dy;
  check(this.{x,y});
 }

 static void clear(int[] a, int n) {
  for (int i = 0; i < n; i++) {
   check(a[i]); a[i]=0;
  }
  check(a[0..n-1]);
 }
}
```

(The `check(this.x);`, `check(this.y);` and `check(a[i]);` calls are struck through; `check(this.{x,y});` and `check(a[0..n-1]);` are highlighted in red.)



Precise Check Placement

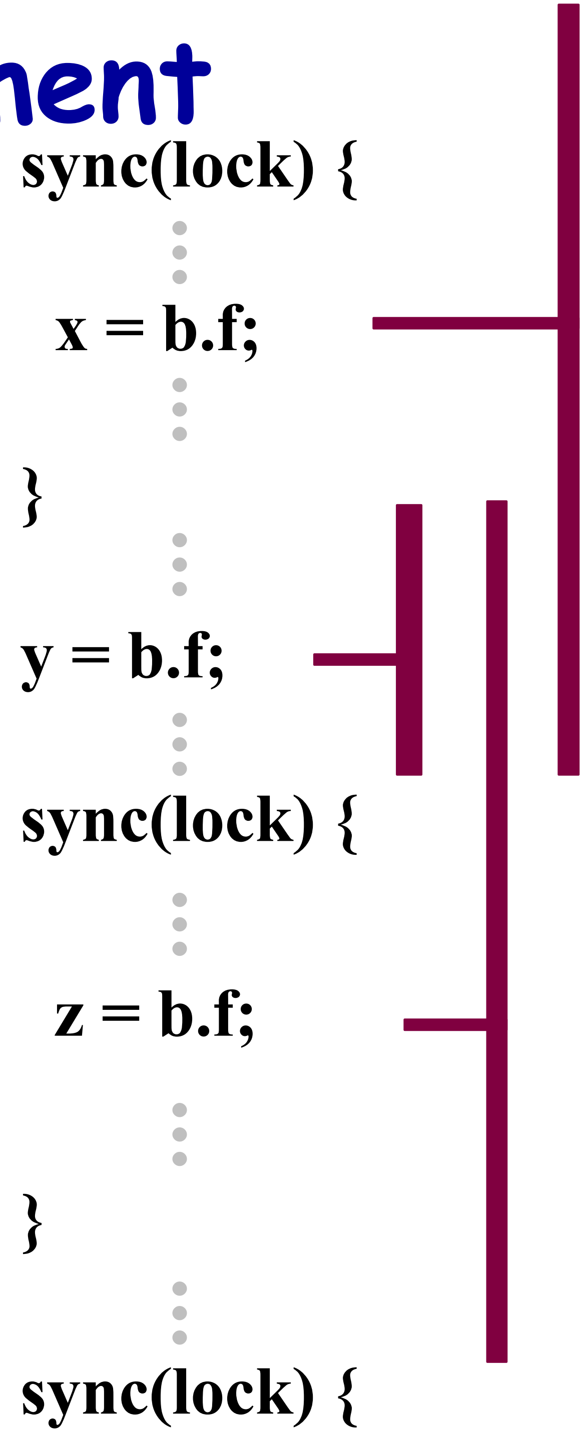
- No Missed Races
- No False Alarms

```
sync(lock) {
    check(b.f)
  x = b.f;
    check(b.f)
}
  .
check(b.f)
y = b.f;
check(b.f)
sync(lock) {
  .
    check(b.f)
  z = b.f;
  .
  .
  .
}
  .
  .
  .
sync(lock) {
```

Precise Check Placement

- **No Missed Races**
- Access must have a covering check between
  - previous release
  - next acquire

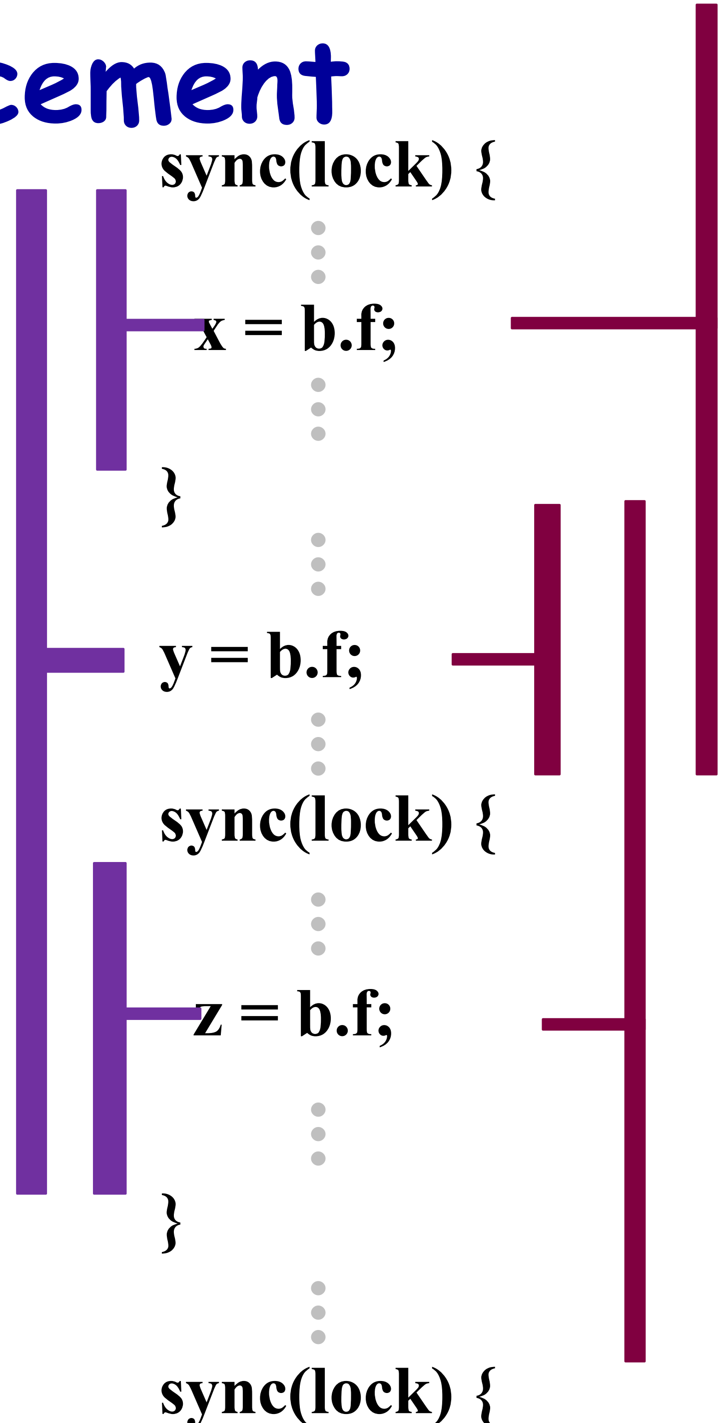
- **No False Alarms**



Precise Check Placement

- **No Missed Races**
- Access must have a covering check between
  - previous release
  - next acquire

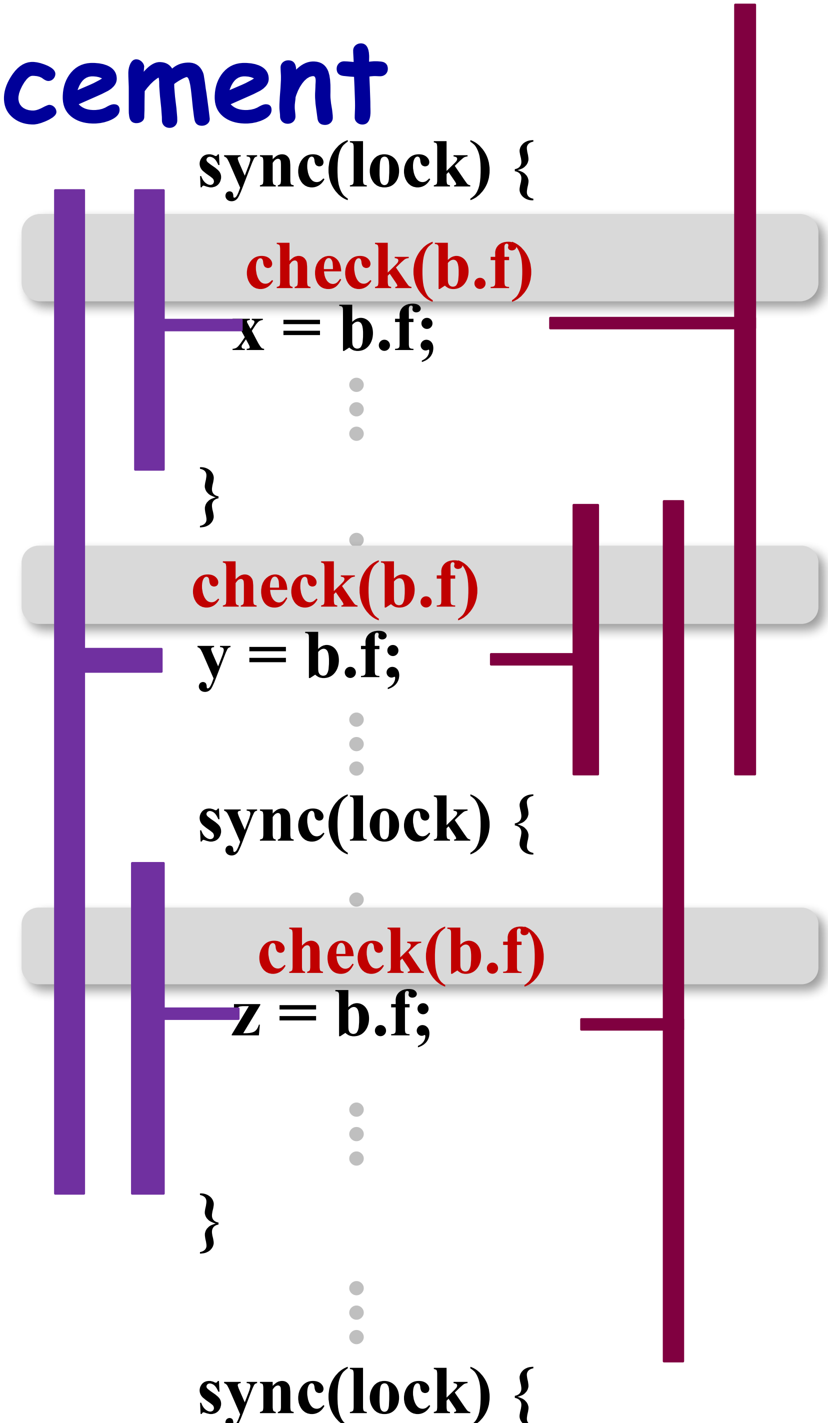
- **No False Alarms**
- Check must have a legitimizing access between
  - previous acquire
  - next release



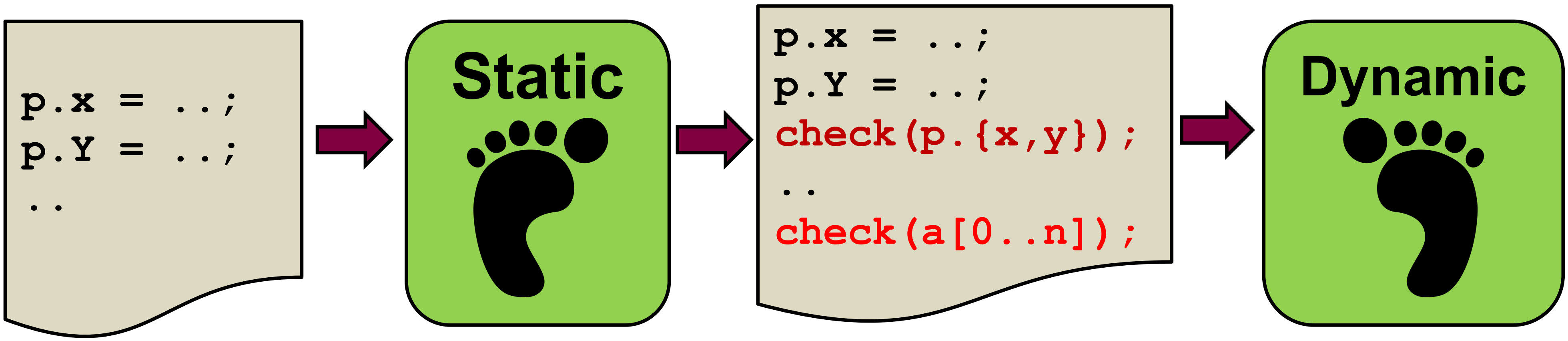
Precise Check Placement

- **No Missed Races**
- Access must have a covering check between
  - previous release
  - next acquire

- **No False Alarms**
- Check must have a legitimizing access between
  - previous acquire
  - next release



BigFoot Overview



1. Static BigFoot
   – Fewer, bigger checks: check(a[0..n])
   – Intra-procedural dataflow analysis
   – WALA [IBM], Z3 [DB 08]
2. Dynamic BigFoot
   – Compress shadow state

Check Placement: FastTrack

```
class Point {
 int x,y;
 void move(int dx, int dy) {
  int tmp;
  check(this.x); tmp = this.x;
  check(this.x); this.x = tmp + dx;
  check(this.y); tmp = this.y;
  check(this.y); this.y = tmp + dy;
 }

 static void clear(int[] a, int n) {
  for (int i = 0; i < n; i++) {
   check(a[i]); a[i]=0;
  }
 }
}
```

Overhead on Benchmarks: 7.3x

Check Placement: BigFoot

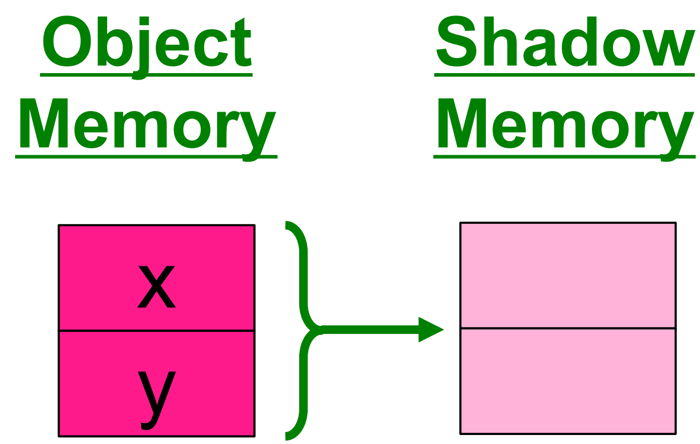
```
class Point {
 int x,y;
 void move(int dx, int dy) {
  int tmp;
  check(this.x); tmp = this.x;
  check(this.x); this.x = tmp + dx;
  check(this.y); tmp = this.y;
  check(this.y); this.y = tmp + dy;
  check(this.{x,y});
 }

 static void clear(int[] a, int n) {
  for (int i = 0; i < n; i++) {
   check(a[i]); a[i]=0;
  }
  check(a[0..n-1]);
 }
}
```

Overhead on Benchmarks: 2.5x

Static Object Shadow Compression

- Compress fields of class that always appear in check statements together

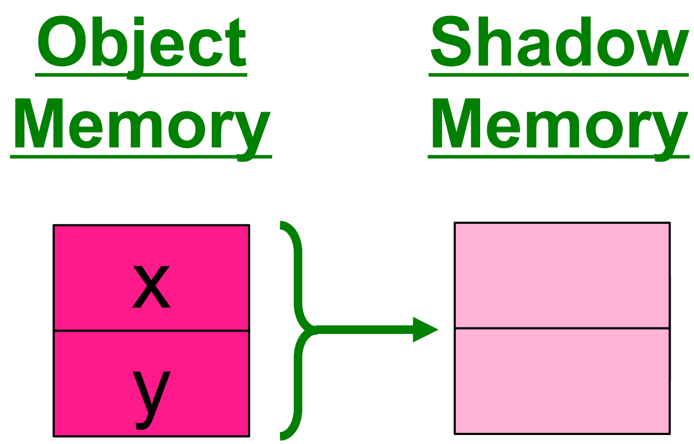
```
...
sync(lock) {
  pt.x = 1;
  check(pt.x);
  pt.y = 2;
  check(pt.y);
}
...
```



Static Object Shadow Compression

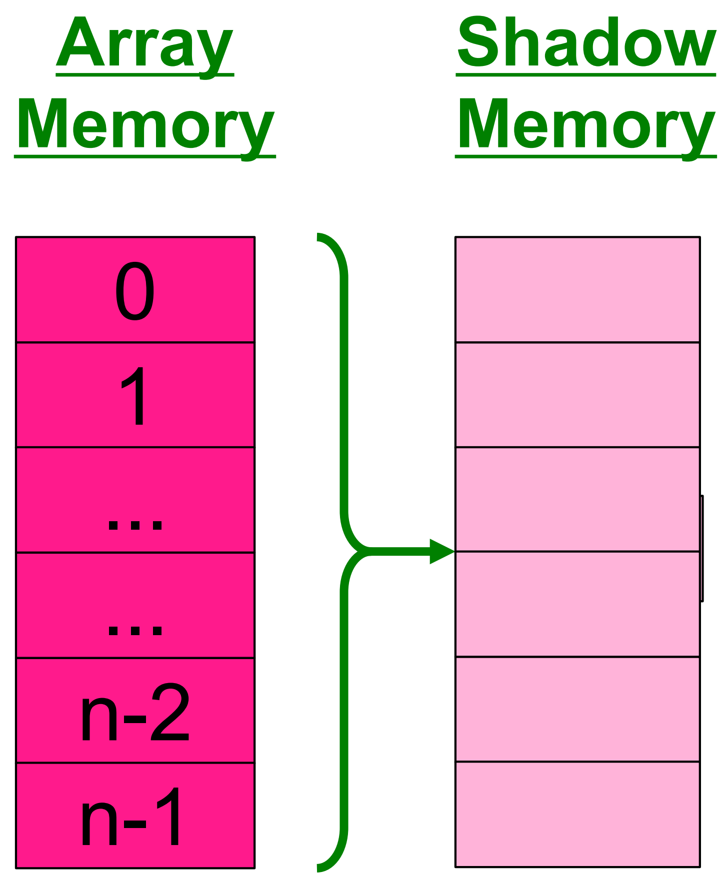
- Compress fields of class that always appear in check statements together

```
...
sync(lock) {
  pt.x = 1;
  pt.y = 2;
  check(pt.{x,y});
}
...
check(b.{x,y});
...
check(c.{x,y});
...
```



Dynamic Array Shadow Compression

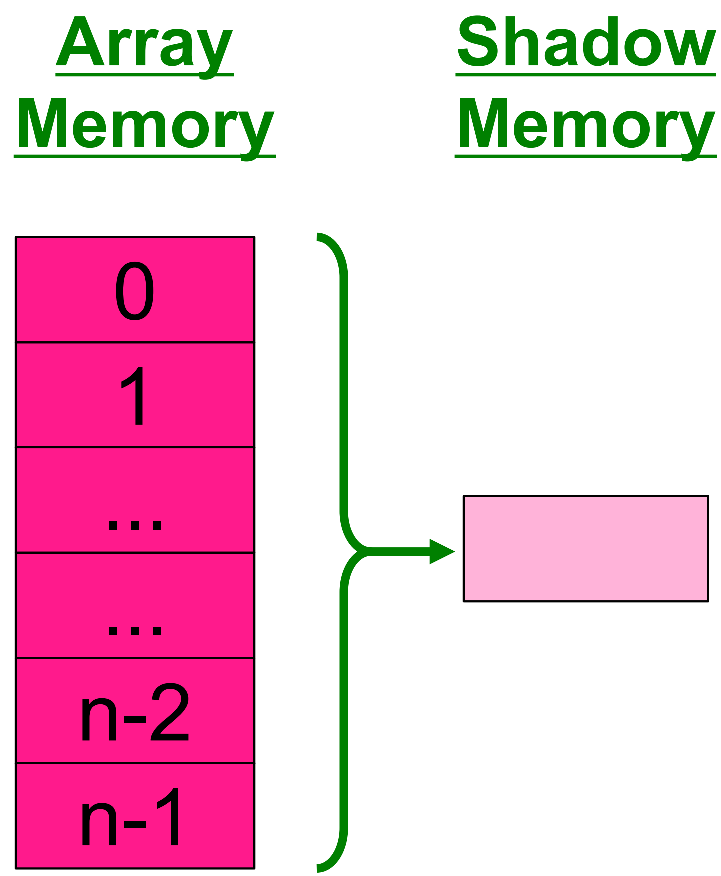
- Initially compress array shadow to single location
- Refine as necessary



Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary

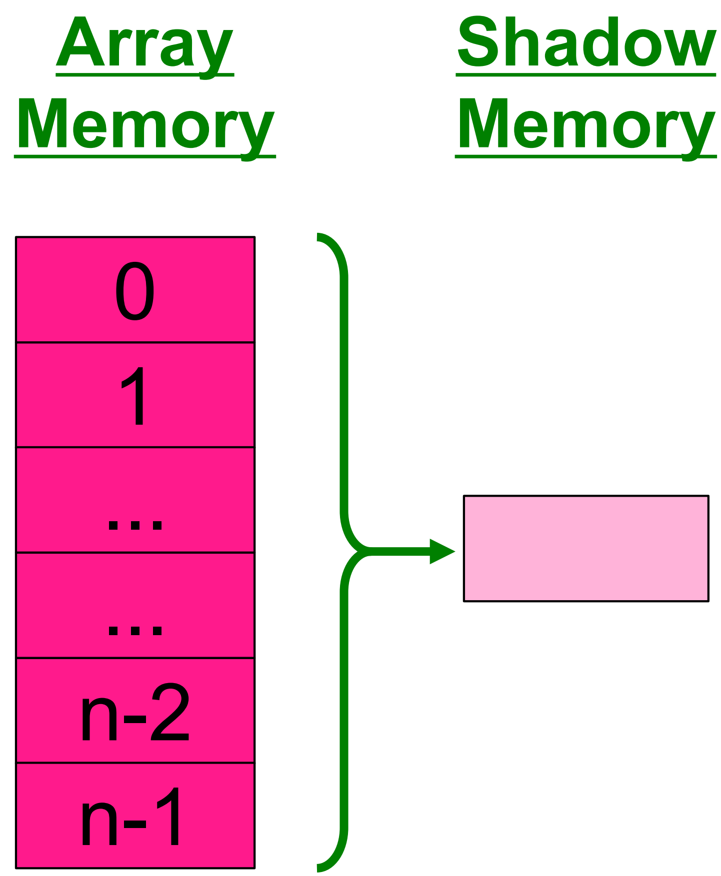
```
sync(lock) {
    for(int i=0;i<n;i++) a[i]=0;
    check(a[0..n-1]);
}
```



Dynamic Array Shadow Compression

- Initially compress array shadow to single location
- Refine as necessary

```
sync(lock) {
    for(int i=0;i<n/2;i++) a[i]=0;
    check(a[0..n/2-1]);
    for(int i=n/2;i<n;i++) a[i]=0;
    check(a[n/2..n-1]);
}
```

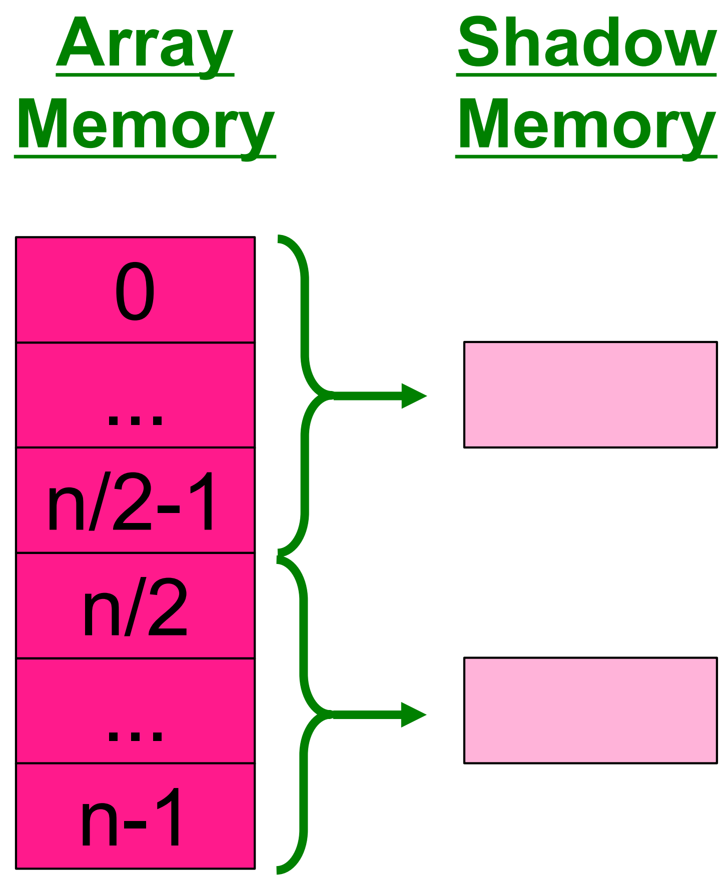


- Buffer checks until release, as in RecPlay [RB 99], DRD [D 14], ThreadSanitizer [SI 09]

Dynamic Array Shadow Compression

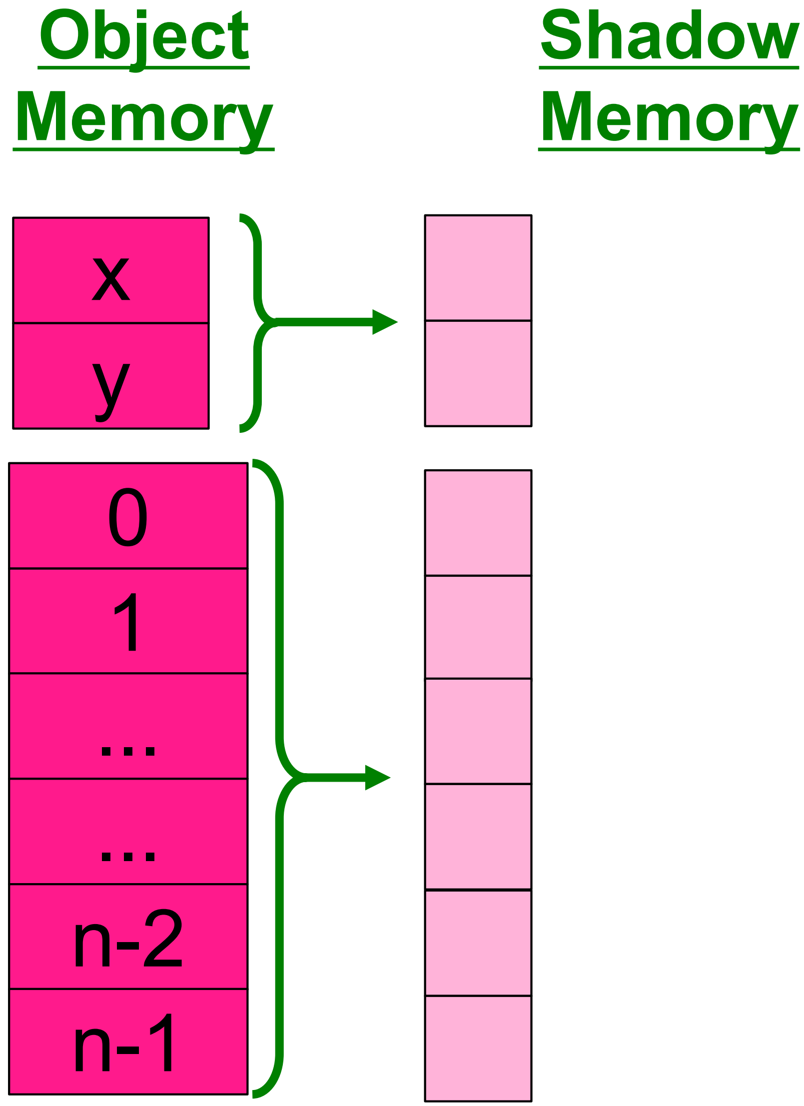
- Initially compress array shadow to single location
- Refine as necessary

```
sync(lock) {
    for(int i=0;i<n/2;i++) a[i]=0;
    check(a[0..n/2-1]);
}
```



Dynamic Race Detection Overhead

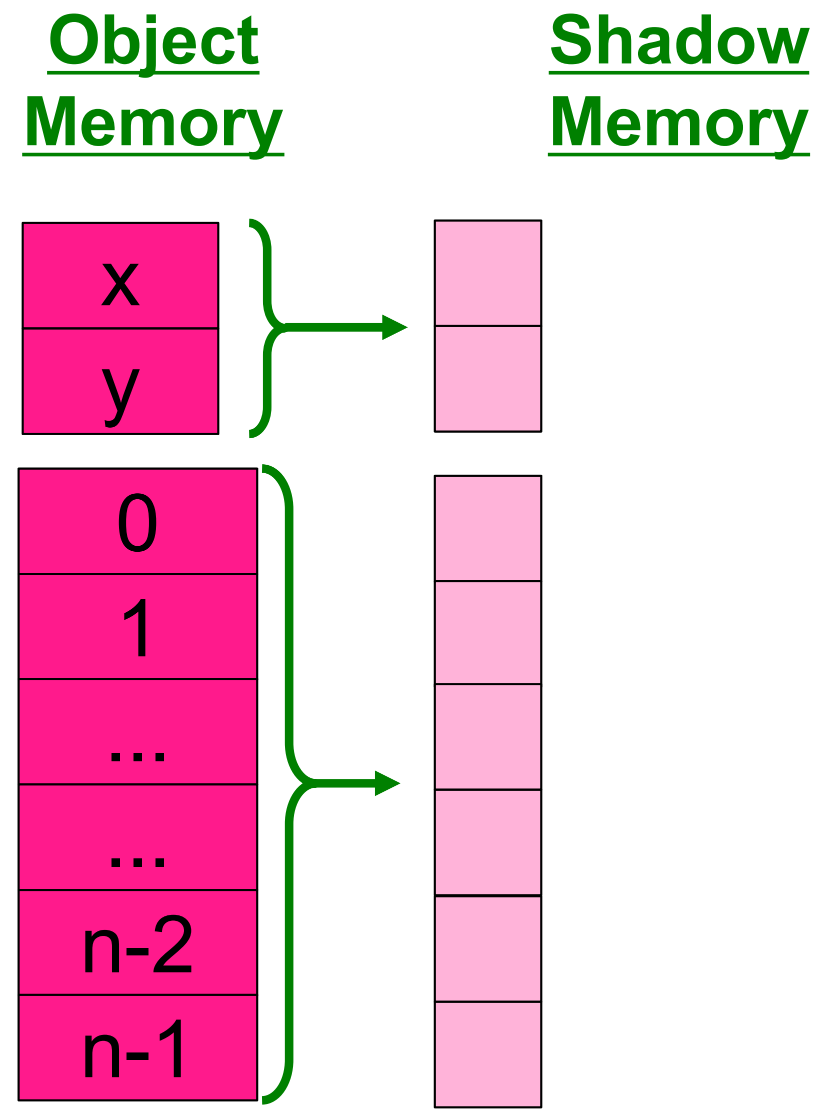
```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    check(this.x); tmp = this.x;
    check(this.x); this.x = tmp + dx;
    check(this.y); tmp = this.y;
    check(this.y); this.y = tmp + dy;
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      check(a[i]); a[i]=0;
    }
  }
}
```



Dynamic Race Detection Overhead

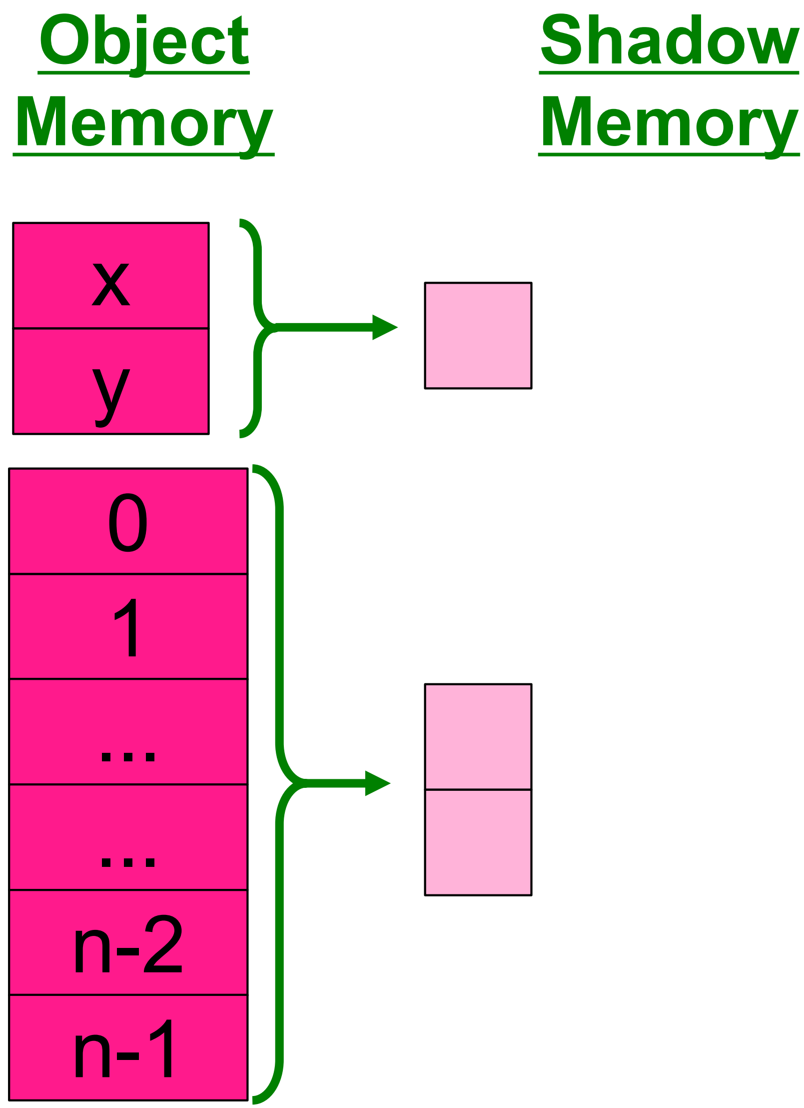
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    ~~check(this.x);~~ tmp = this.x;
    ~~check(this.x);~~ this.x = tmp + dx;
    ~~check(this.y);~~ tmp = this.y;
    ~~check(this.y);~~ this.y = tmp + dy;
    check(this.{x,y});
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      ~~check(a[i]);~~ a[i]=0;
    }
    check(a[0..n-1]);
  }
}

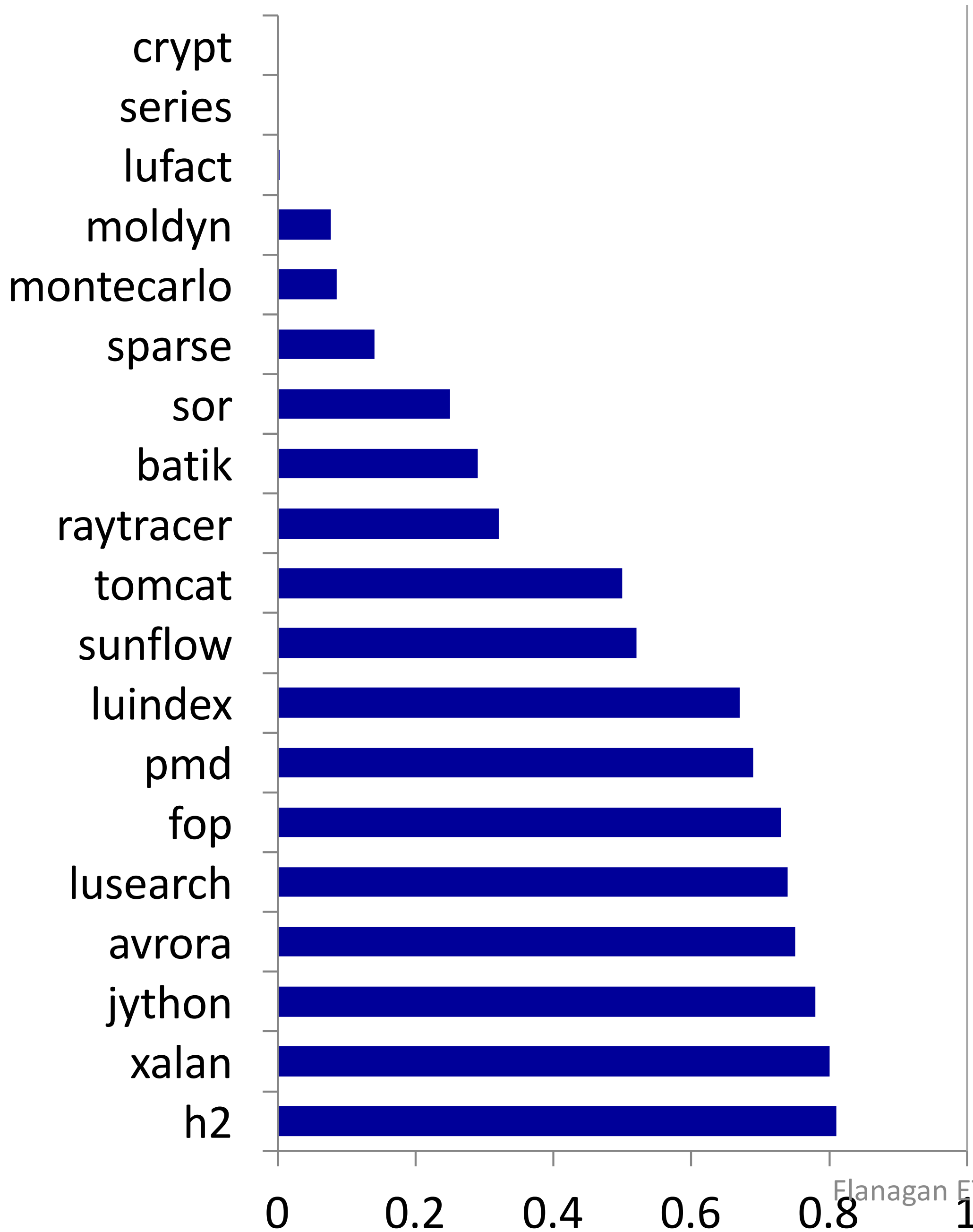


Dynamic Race Detection Overhead

```
class Point {
  int x,y;
  void move(int dx, int dy) {
    int tmp;
    ~~check(this.x);~~ tmp = this.x;
    ~~check(this.x);~~ this.x = tmp + dx;
    ~~check(this.y);~~ tmp = this.y;
    ~~check(this.y);~~ this.y = tmp + dy;
    check(this.{x,y});
  }

  static void clear(int[] a, int n) {
    for (int i = 0; i < n; i++) {
      ~~check(a[i]);~~ a[i]=0;
    }
    check(a[0..n-1]);
  }
}
```



BigFoot Eliminates Checks

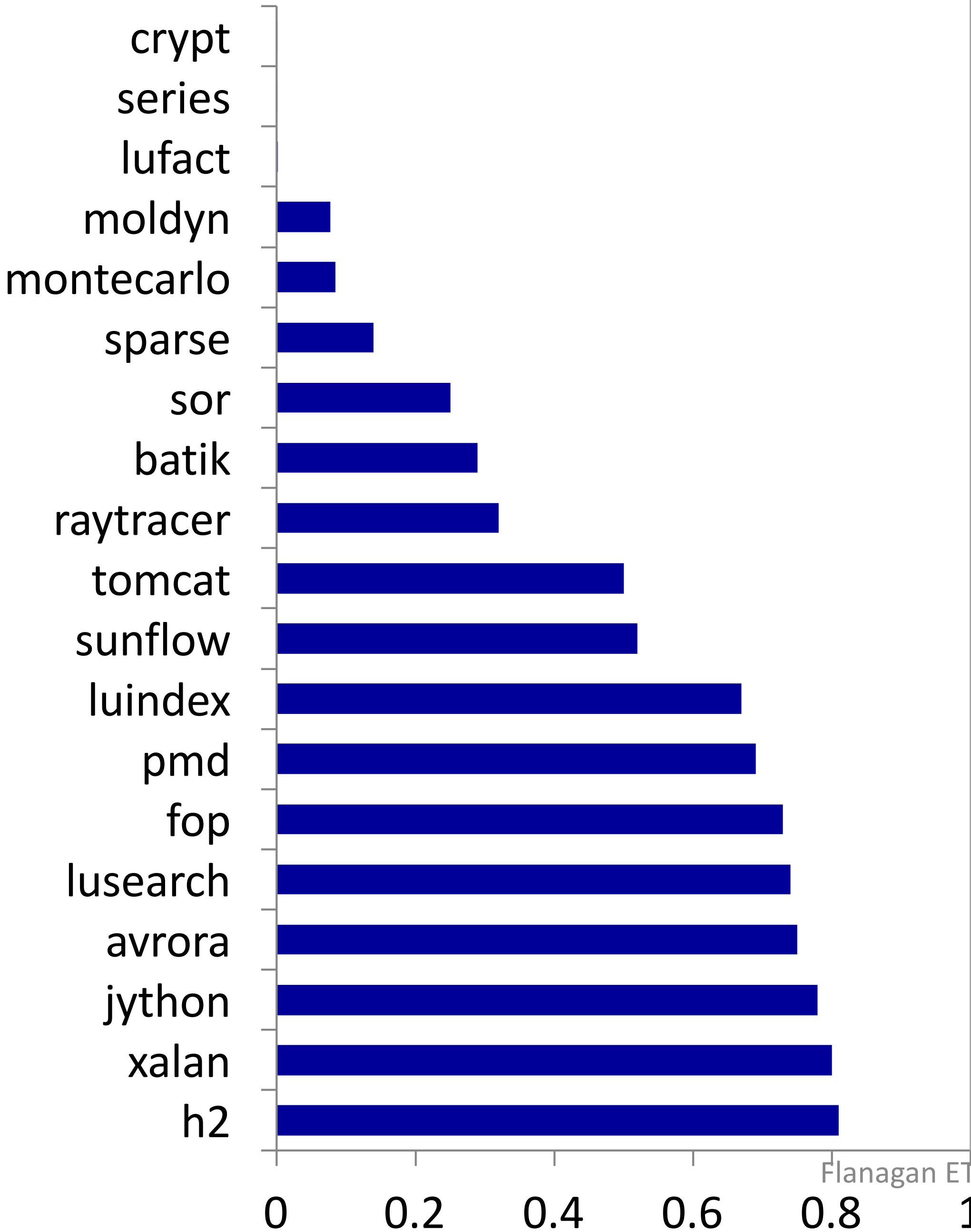


Check Ratio = $\frac{\text{\# Checks}}{\text{\# Accesses}}$

Lower is Better

- FastTrack: 1
- BigFoot: 0.43

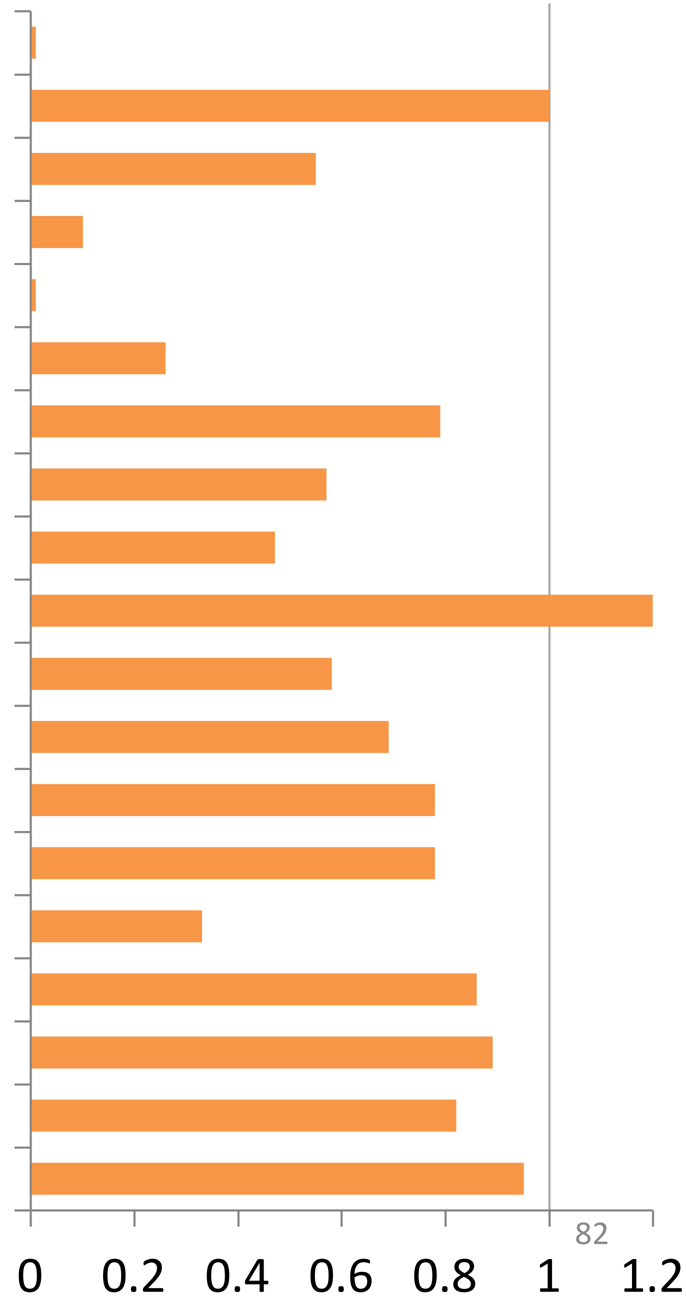
BigFoot Check Ratio



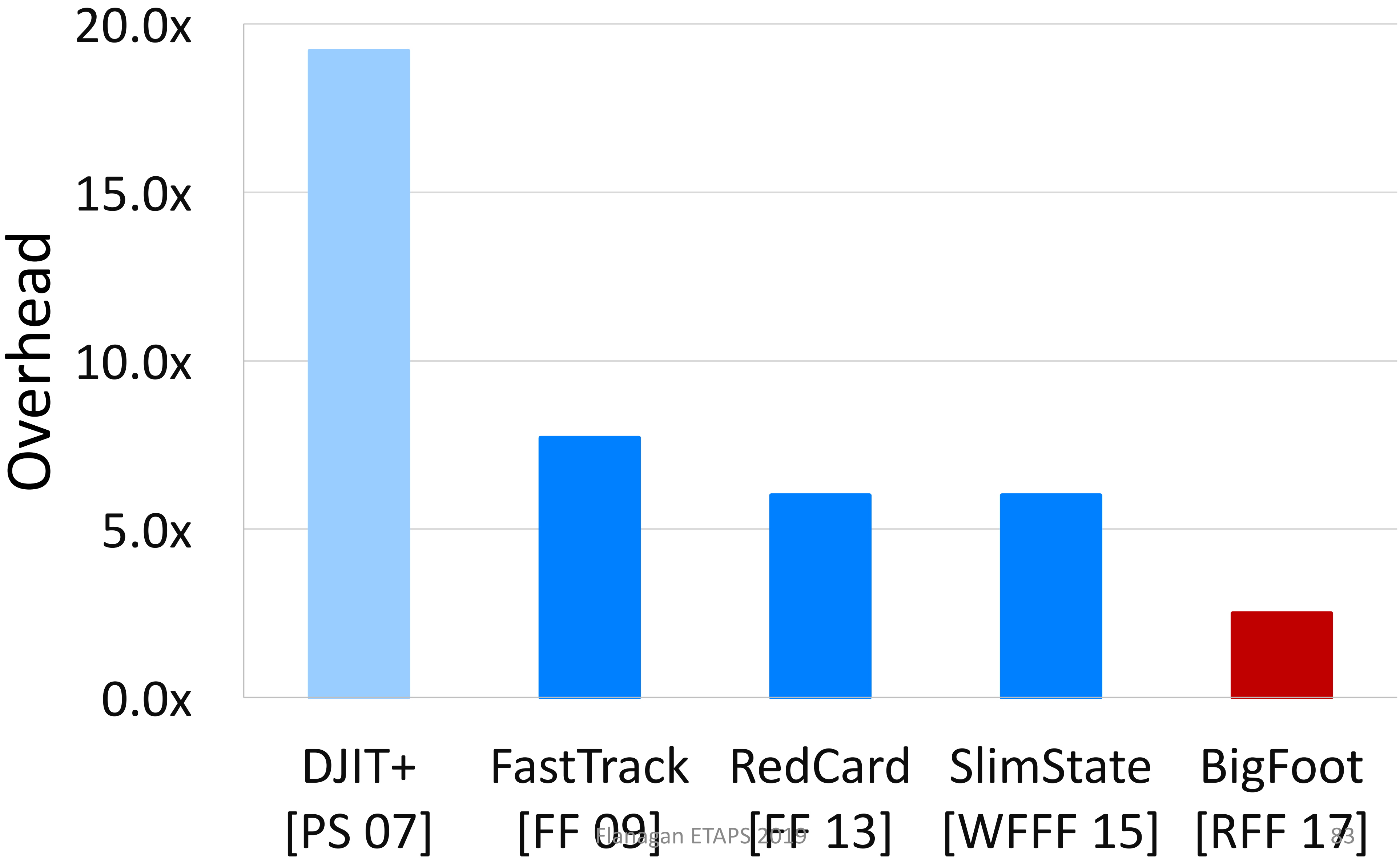
Overhead

Benchmark	FT	BF
crypt	95.2	0.1
series	0.0	0.0
lufact	71.4	39.5
moldyn	27.5	2.7
montecarlo	7.4	0.1
sparse	26.7	6.8
sor	13.9	11.0
batik	4.0	2.3
raytracer	13.5	6.3
tomcat	2.0	2.4
sunflow	25.9	15.1
luindex	16.4	11.3
pmd	3.1	2.4
fop	6.5	5.1
lusearch	19.5	6.5
avrora	1.4	1.2
jython	9.3	8.3
xalan	5.6	4.6
h2	3.2	3.0

Ratio



Precise Dynamic Race Detection



Summary

- Race freedom - **as if** sequentially consistent
- Atomic methods - **as if** executed serially
- Explicit yields - **as if** on non-preemptive scheduler