Location Pairs: A Test Coverage Metric for Shared Memory Concurrent Programs

Serdar Tasiran, M. Erkan Keremoğlu, Kıvanç Muslu

Koç University
Istanbul, Turkey

The Location-Pairs Coverage Metric

- A test coverage metric for shared-memory concurrent programs
  - Have I explored enough interesting and distinct thread interleavings during testing?
  - What other interleavings should I try to exercise?

- Corresponds well to atomicity and refinement violations

- A good compromise between complexity and bug detection ability

Location Pairs and Atomicity Violations

```
class StringBuffer {

  /**
  *  Used for character storage.
  */
  char[] value;


  /**
  *   number of valid characters in “value”
  */
  int count;
}
```

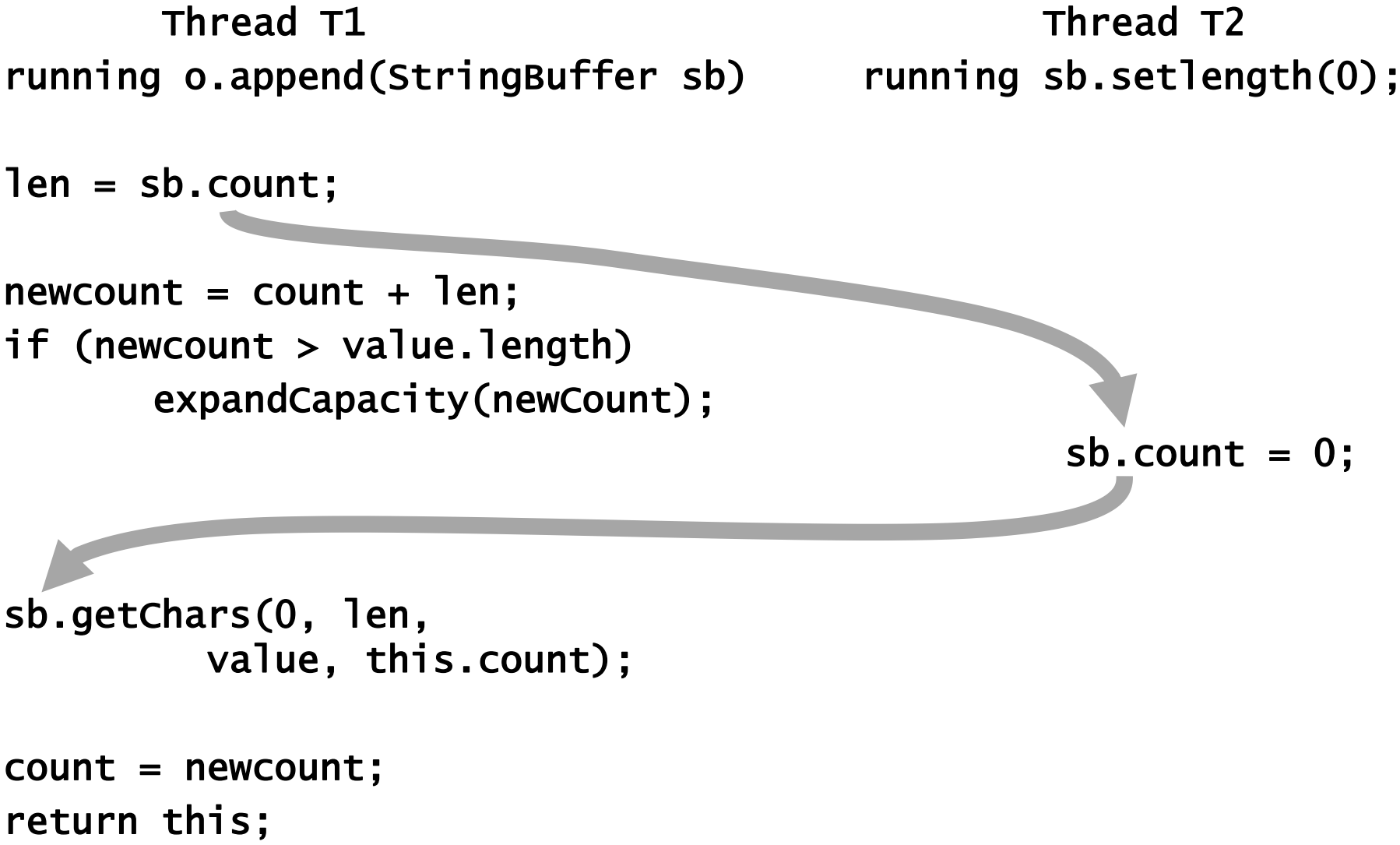
Location Pairs and Atomicity Violations

```
        Thread T1
running o.append(StringBuffer sb)

len = sb.count;

newcount = count + len;
if (newcount > value.length)
        expandCapacity(newCount);




sb.getChars(0, len,
          value, this.count);

count = newcount;
return this;
```

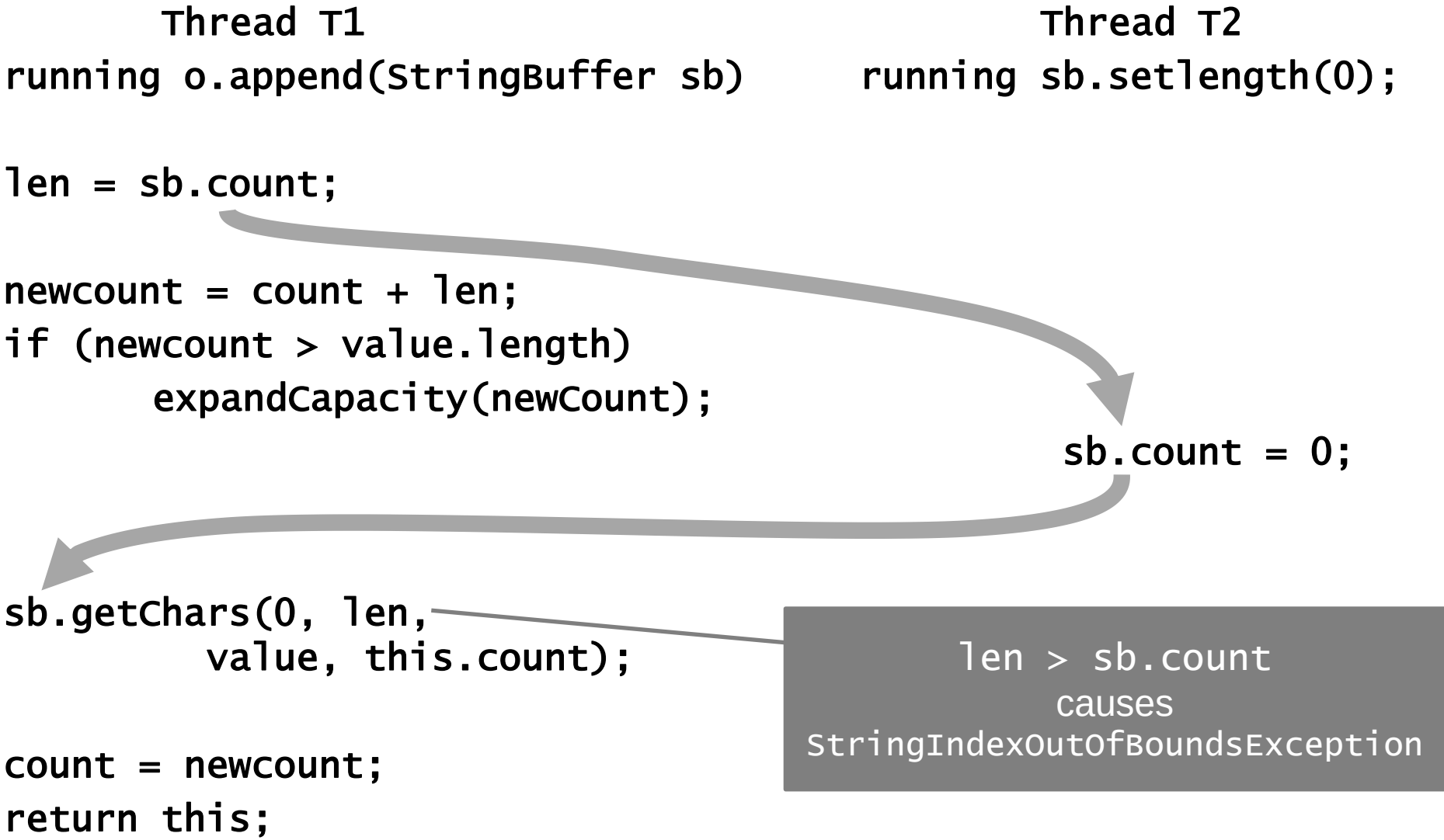
Location Pairs and Atomicity Violations

```
        Thread T1                                   Thread T2
running o.append(StringBuffer sb)        running sb.setlength(0);

len = sb.count;

newcount = count + len;
if (newcount > value.length)
        expandCapacity(newCount);
                                                    sb.count = 0;



sb.getChars(0, len,
          value, this.count);

count = newcount;
return this;
```

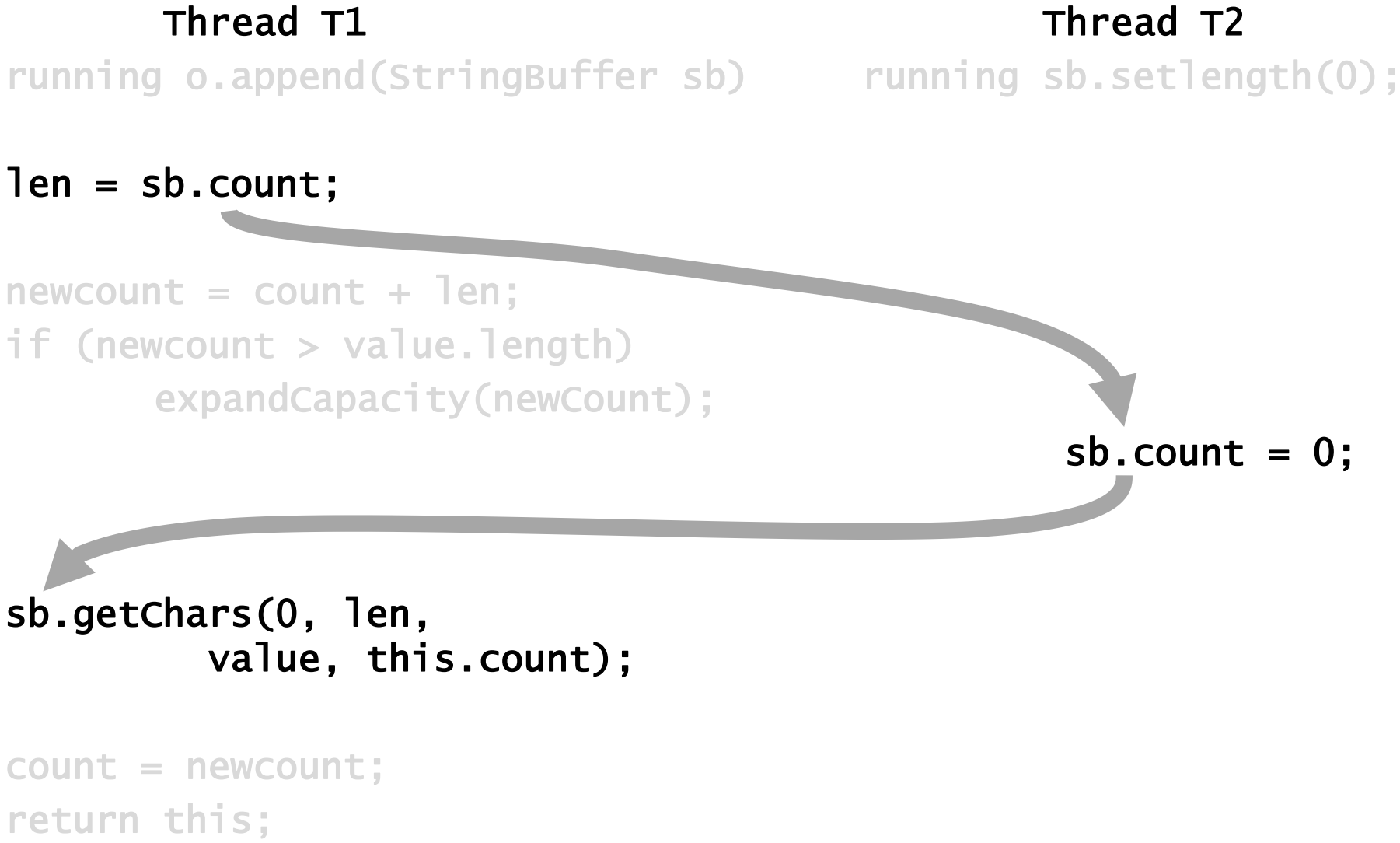
Location Pairs and Atomicity Violations



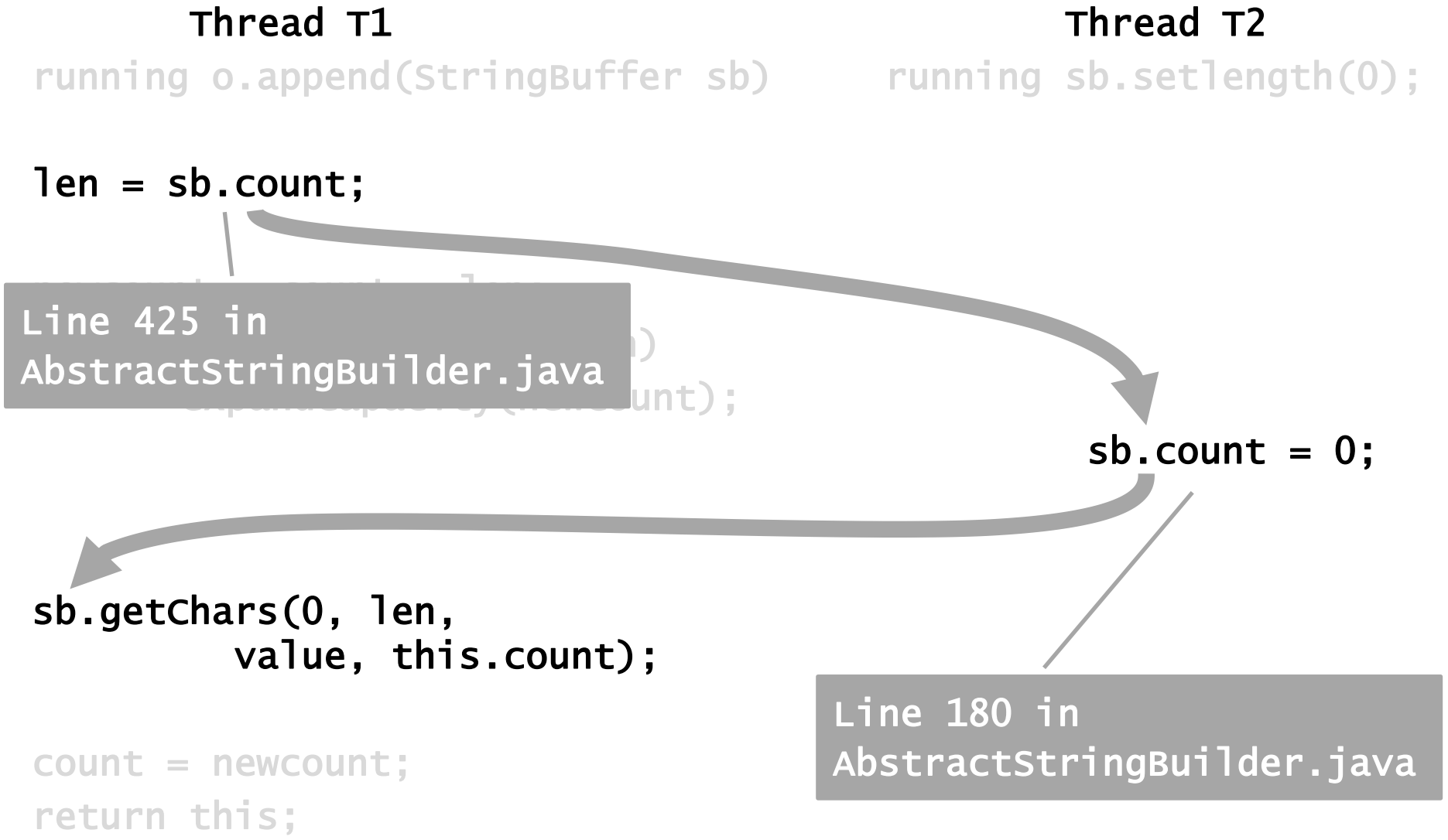
Location Pairs and Atomicity Violations



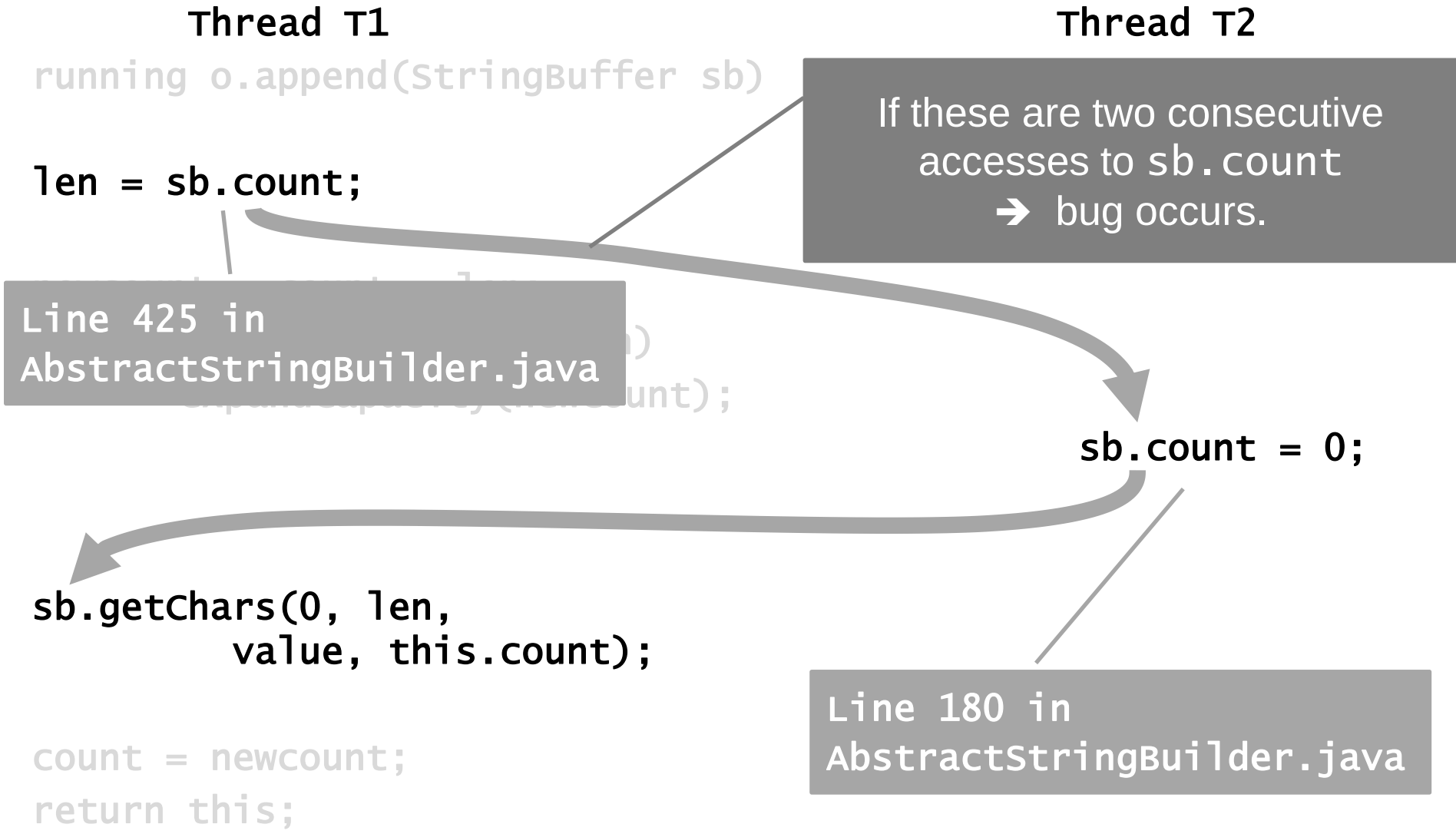
Pattern Causing Atomicity Violation



Location Pairs



Location Pairs



All Definitions-Uses vs Location Pairs

Thread T1

```
running o.append(StringBuffer sb)

len = sb.count;

newcount = count + len;
if (newcount > value.length)
        expandCapacity(newCount);

sb.getChars(0, len,
          value, this.count);

count = newcount;
return this;
```

Use

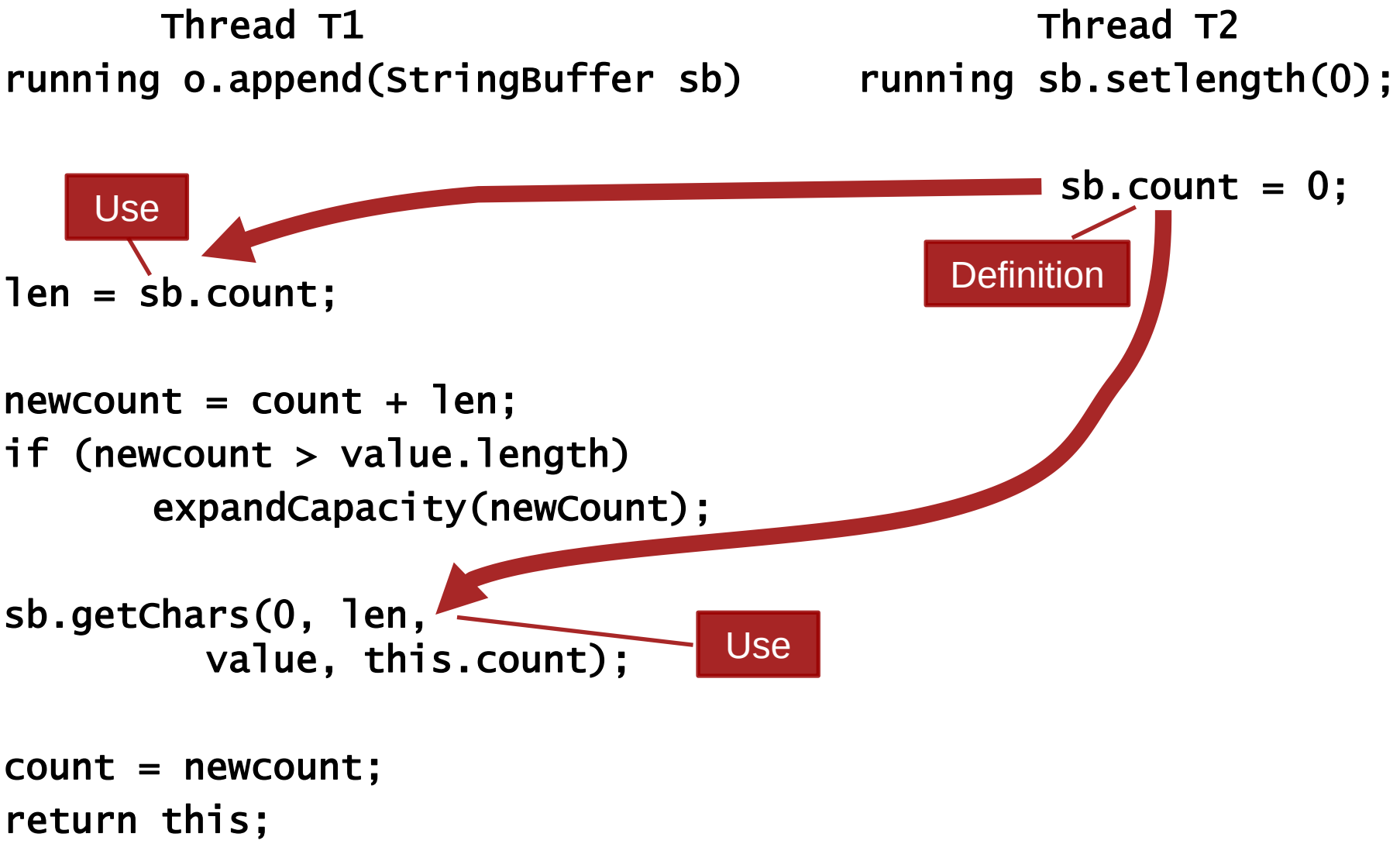
Use

Thread T2

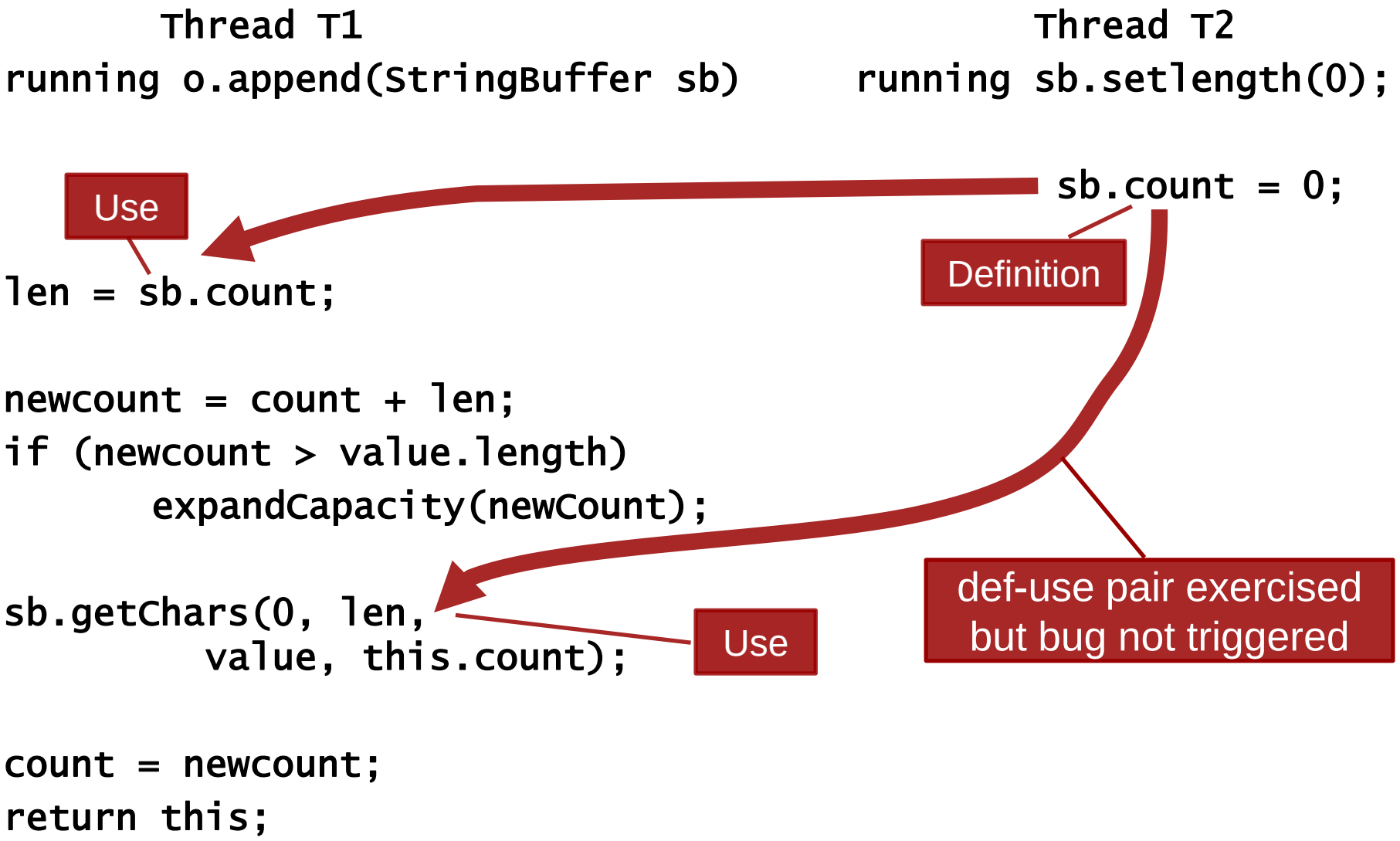
```
running sb.setlength(0);

sb.count = 0;
```

Definition

All Definitions-Uses vs Location Pairs



All Definitions-Uses vs Location Pairs



Inspiration for the LP Metric

- Based on bugs in the following studies, most captured by LP
  - “Learning from mistakes: A comprehensive study of real world concurrency bug characteristics”
    Lu, Park, Seo, Zhou, ASPLOS ‘08
  - “A study of interleaving coverage criteria”
    Lu, Zhiang, Zhou, FSE ‘07
  - “Verifying concurrent programs by runtime refinement-violation detection”
    Elmas, Qadeer, Tasiran, PLDI ’05
  - “Types for atomicity”
    Flanagan, Qadeer, TLDI’ 03
  - “Concurrent bug patterns and how to test them”
    Farchi, Nir, Ur, IDPDS ’03

Concurrent Coverage Metrics Issues

- Have I tested enough?
- Is my current way of testing still helping?
- Where else should I focus my testing effort?
  - What inputs and interleavings should I prioritize?
- Is this metric a good proxy for the bugs I am after?
  - If I achieve 100% coverage, am I guaranteed/likely to catch all errors in a certain category?
- How hard is it to accomplish coverage with respect to this metric?
  - Is the coverage target of reasonable size?
  - Is the coverage gap remaining after lots of random testing small enough to tackle manually?

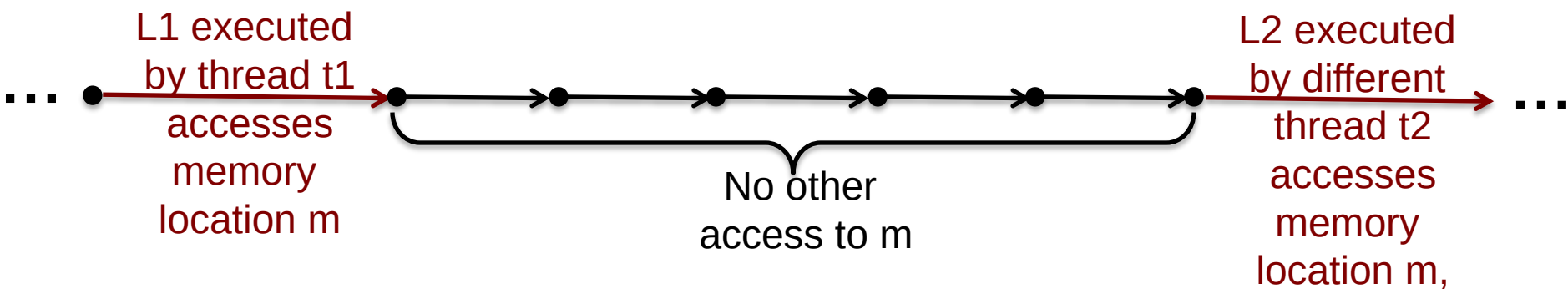
The Rest of the Talk

- Location pairs: formal definition
- Static computation of coverage target
- Coverage measurement tool implementation
- Experiments: Comparison
  - Bug detection ability
  - Saturation experiments
- Interactive debugging example

Location Pair Coverage

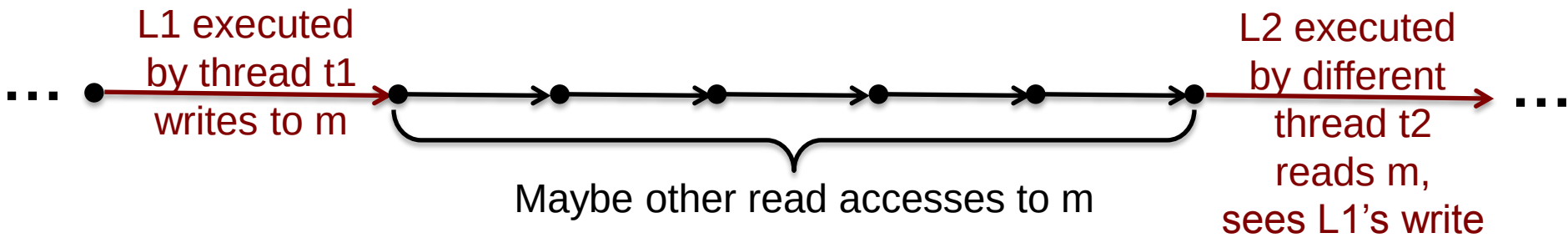
- Location pair: (L1, L2)
  - (L1,L2): A pair of bytecode instructions in the program
    - L1 can be the same as L2
  - At least one of L1 and L2 is a write access

- (L1,L2) covered by execution if



Other, similar metrics

- All concurrent definition-use pairs (DU)
  - L1, the “definition”: a write to a variable m
  - L2, the “use”: a read of variable m
- DU pair (L1,L2) exercised iff



- Method pairs (MP)
  - M1, M2: Methods
  - (M1,M2) covered if
    - an action from M2 is executed by a different thread while M1 is in progress

Coverage Measurement Tool

- Implemented as Java PathFinder VMListener
  - JPF notifies tool after every bytecode instruction
- JPF explores thread interleavings

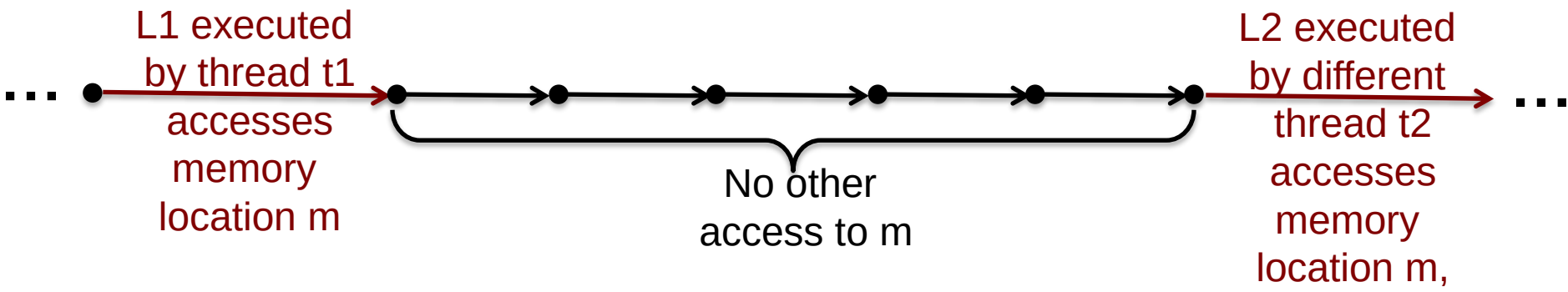
- Coverage tool not meant to be efficient
- Goal: ShowLP metric corresponds well with concurrency bugs

- Tool issues:
  - JPF storing explored states → **A lot** of space
  - Even sequence of states leading to current state is a bottleneck

- Solution:
  - Modify JPF not to store any states
  - Use it as runtime instrumentation engine with scheduling control.

Coverage Target

- When do we have 100% coverage?
  - When we cover all coverable (L1, L2)



- Static analysis to determine/approximate coverable pairs
- Is (L1,L2) coverable?
  - Closely related to (L1,L2) being involved in a race condition
  - Difference: Even when there is proper synchronization between L1 and L2, (L1,L2) may be coverable.
- We make use of analyses in the Chord static race detection tool.

Coverable Pairs

- Chord [Naik et al, PLDI '06]

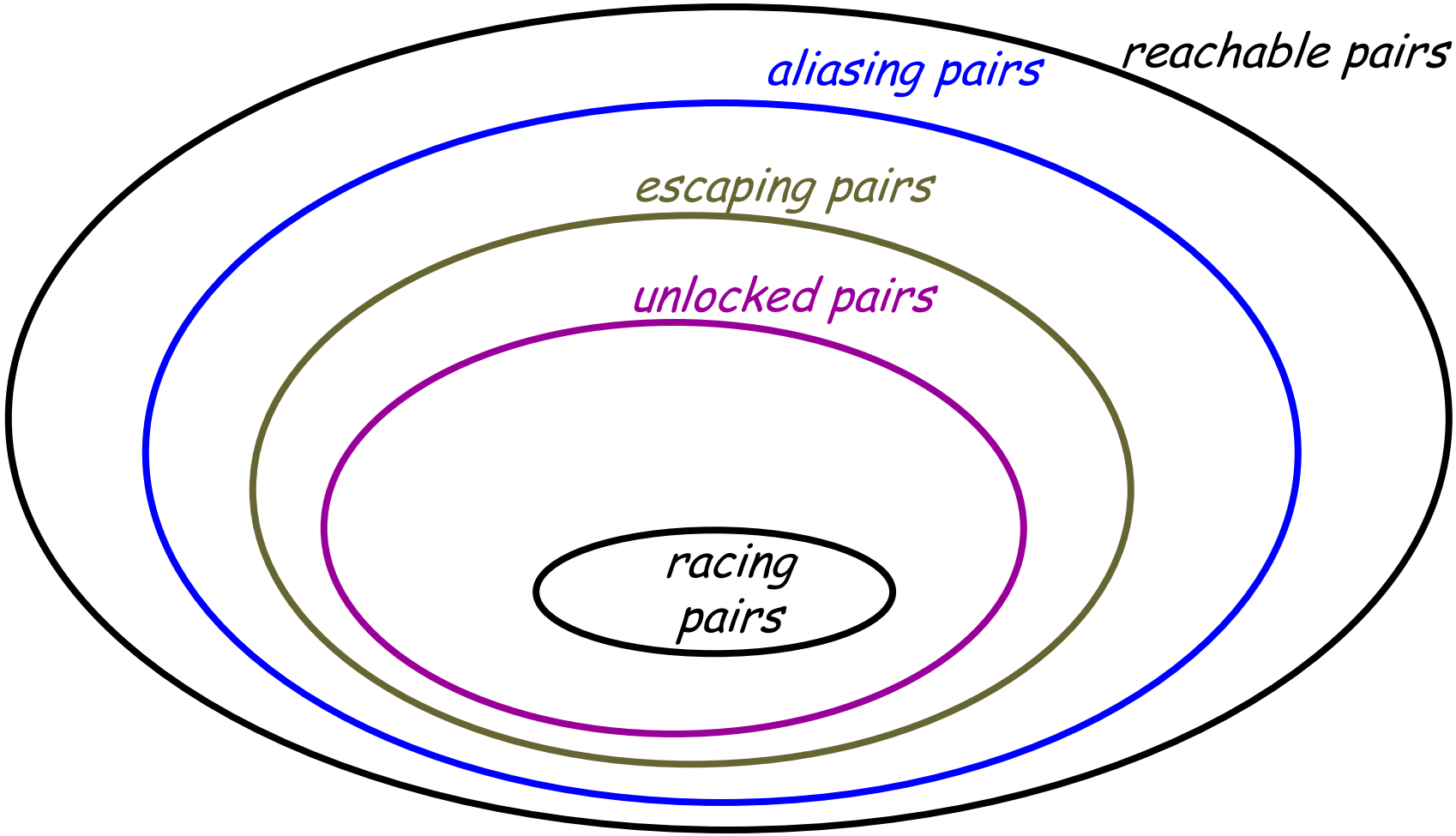


Figure courtesy of Naik et al.

Coverable Pairs

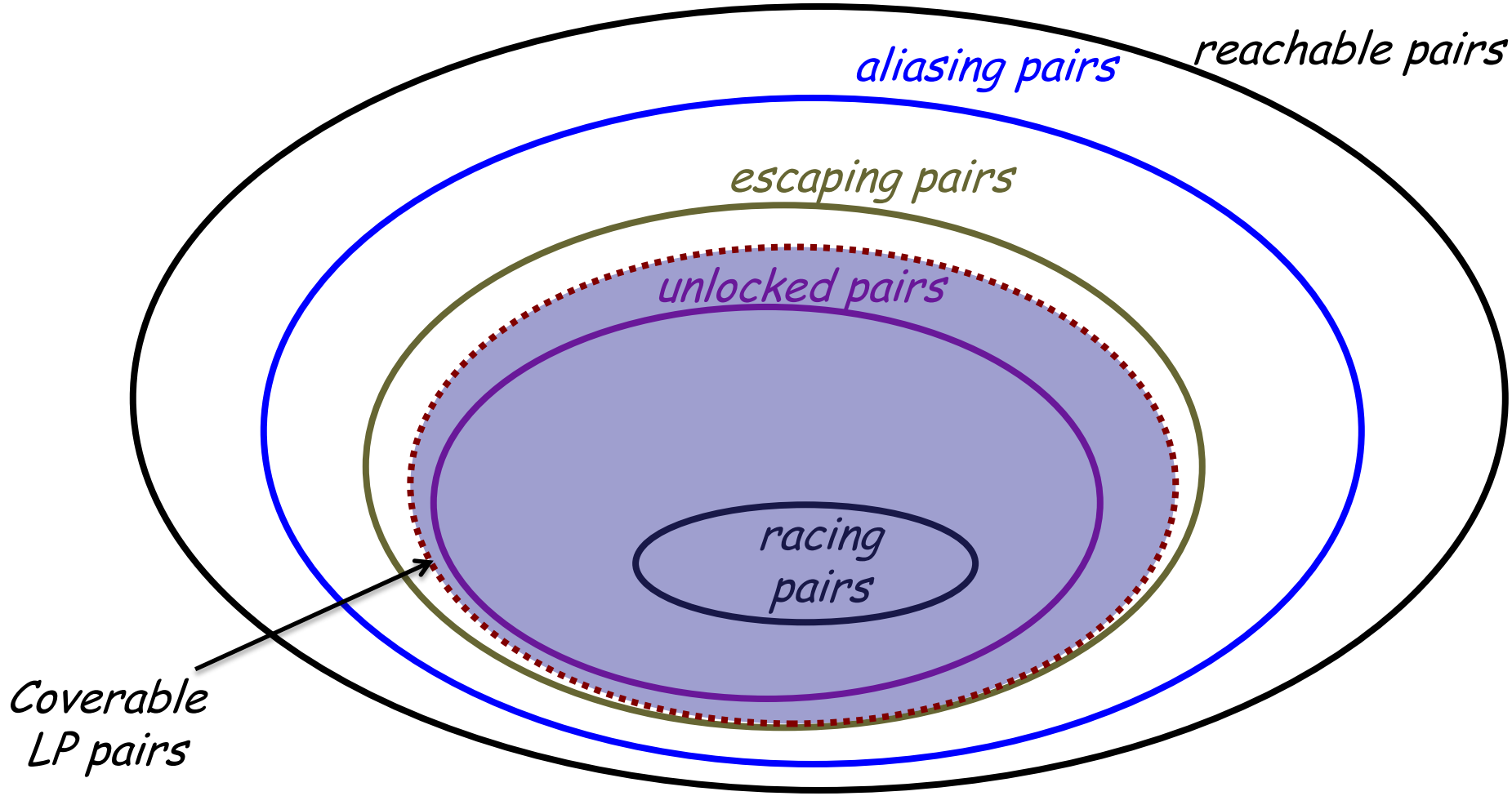


Figure courtesy of Naik et al.

Coverable Pairs: Static Overapproximation

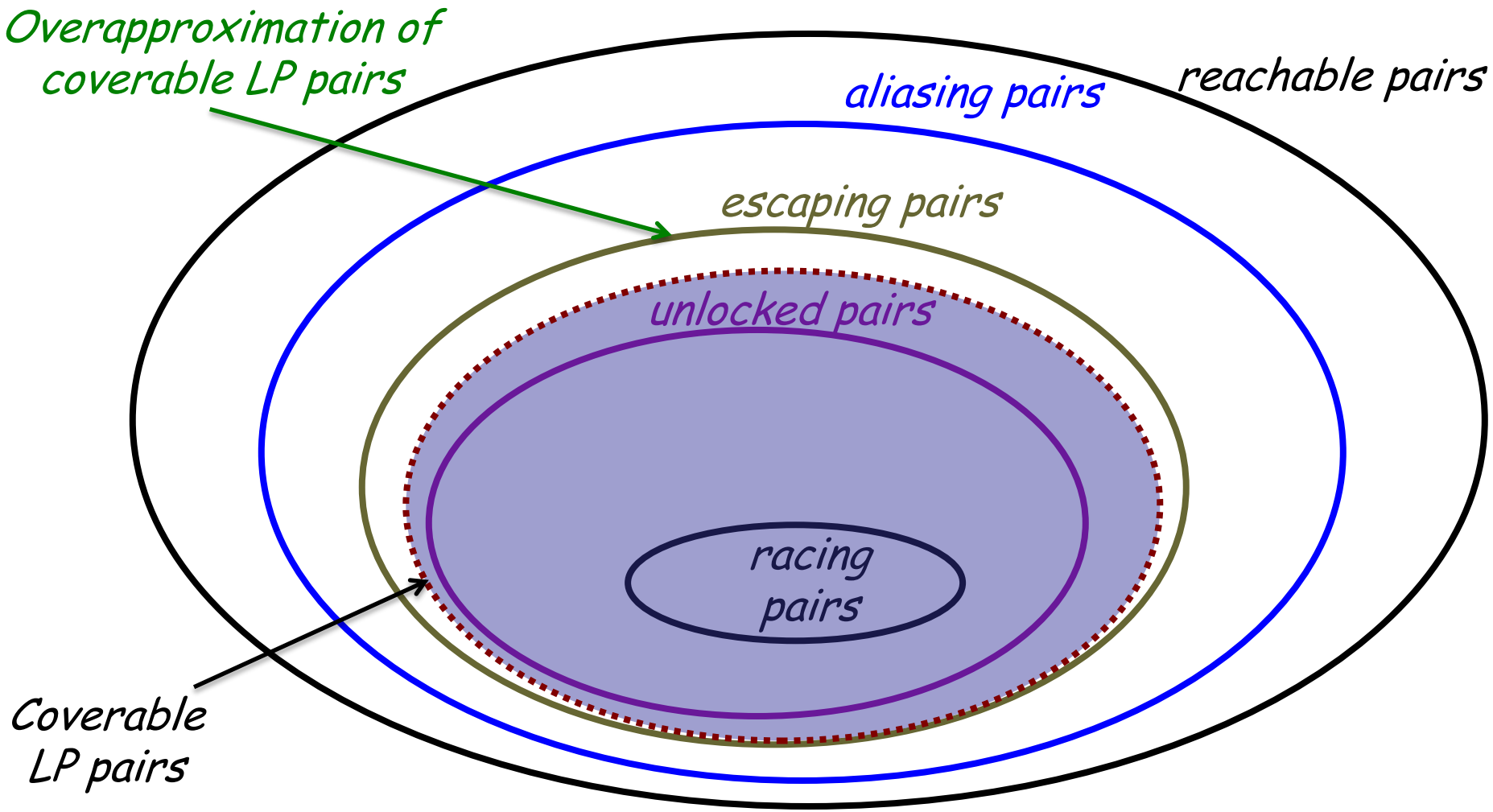


Figure courtesy of Naik et al.

- Wait!
  What if the approximate coverage target is huge?
  - It isn't.

Program	Size of SP set	Size of program (Loc)
Knapsack	45	136
Eventdrivensimulation	246	2,148
Prim	84	1,450
Delaunayrefinement	1,191	1,317
Elevator	697	358
BarnesHut	467	473
PhylogenyParsBnbSmpFixed	871	6,978
PhylogenyParsBnbSmp	871	6,978
FindKeySmp3	384	5,963
FloydSmpCol	106	5,508
FloydSmpRow	106	5,508
PiSmp3	114	7,593
PiSmpFixed	114	7,593
PiSmp	114	7,593
MandelbrotSetSmp	103	5,850
Tsp	587	450
JGFSORBenchSizeA	44	238
JGFSeriesBenchSizeA	14	225
JGFCryptBenchSizeA	316	358
JGFLUFactBenchSizeA	83	559
JGFSparseMatmultBenchSizeA	19	220
JGFMonteCarloBenchSizeA	55	1,260
JGFRayTracerBenchSizeAFixed	549	851
JGFRayTracerBenchSizeA	555	851

LP Coverage: Intended Use

- Static analysis computes overapproximation of coverage target

- Run tests, use randomized scheduling, maybe different input data
  - Use systematic exploration a la Chess if you like

- For each non-covered pair
  - By inspection, rule out as “not coverable,” or
  - Devise scenario to cover it
    - Pick data, hand-craft schedule to cover LP
  - This is where LP helps focus test generation effort

- Feasibility shown on benchmarks:
  - Moldyn, SOR, TSP, Prim, Elevator, Multiset, Apache FTPServer

Testing Moldyn

```
358     if(id == 0) {
359       for (j=0;j<3;j++) {
360         for (i=0;i<mdsize;i++) {
361           sh_force[j][i] = sh_force[j][i] * hsq2;
362         }
363       }
364     }
    ...
369      br.DoBarrier(id);
    ...
    ...
    ...
552     xvelocity = xvelocity + sh_force[0][part_id];
553     yvelocity = yvelocity + sh_force[1][part_id];
554     zvelocity = zvelocity + sh_force[2][part_id];
    ...
```

- Static analysis:

  26 coverable pairs
- Initial random testing covers only 9 pairs
- Longer tests cover 23 pairs

- Remaining 3 pairs:
  - (361,552)
  - (361,553)
  - (361,554)
  - Only thread 0 executes 358-364
  - Thread 0 always first to reach 552-554 in experiments

- Pause thread 0, let other threads continue ➔ All pairs covered.

Bug detection ability: Comparison with other metrics

- Create buggy programs
  - Mutation operators for concurrent Java [Bradbury et al. ‘06]
    - SHCR: Shrink Critical Region
    - EXCR: Expand Critical Region
    - SPCR: Split Critical Region
    - MSP: Modify Synchronized Block Parameter
    - RSB: Remove Synchronized Block
    - ESP: Exchange Synchronized Block Parameter
  - Manually inserted atomicity violations
    - Re-order code, move certain reads and writes outside synchronized block
- Bug theme:
  - Code block intended to be atomic
  - But, in fact, split into several atomic blocks

Experimental Comparison Setup

- Definition: One **pass**
  - 2-3 threads performing 2-3 operations each, or
  - One execution of program, from start to finish
- Bug caught by pass: Assertion violated during pass
  - Assertions relevant to bug manually inserted
  - Examples:
    - Assertions about data structure state or matrix contents
    - Assertions about method return values
- Definition: One **iteration**:

```
while (bug not caught)
    perform pass
```

- Measure different kinds of coverage during iteration

Experimental Comparison Setup

- Definition: One iteration:

  while (bug not caught)

  perform pass

- Metric not successful as a measure of test adequacy
  - Metric reaches 100% coverage during an iteration
  - But bug not caught by then

- Other measures of correlation with bugs:

  - % of passes that cover a new LP/MP/DU that also catch the bug

  - % of passes that catch the bug that also cover a new LP/MP/DU

Buggy benchmarks	Success of metric as adequacy measure (%)		
	MP	DU	LP
MS mutant 6	1	6	100
MS mutant 9	2	7	100
MS mutant 14	100	100	100
MS mutant 15	100	100	100
MS + seeded error 1	31	29	100
MS + seeded error 2	7	100	100
MS + seeded error 3	2	22	100
MS + seeded error 4	100	100	100
EL1 mutant	100	100	100
EL2 mutant	100	100	100

Buggy benchmarks	% passes that cover a new MP/DU/LP that also detect bug		
	MP	DU	LP
MS mutant 6	0	0.8	18.5
MS mutant 9	0.2	0.3	17.6
MS mutant 14	0	1.5	6.1
MS mutant 15	0.4	24.7	14.7
MS + seeded error 1	26.6	27.0	60.0
MS + seeded error 2	0.6	22.3	14.1
MS + seeded error 3	0.5	1.4	13.9
MS + seeded error 4	0.3	1.2	12.9
EL1 mutant	12.5	6.1	16.8
EL2 mutant	9.7	5.9	17.4

Buggy benchmarks	% passes that detect bug that cover a new MP/DU/LP		
	MP	DU	LP
MS mutant 6	0	3	100
MS mutant 9	1	1	99
MS mutant 14	0	12	100
MS mutant 15	3	100	100
MS + seeded error 1	30.3	26.4	100
MS + seeded error 2	5	100	100
MS + seeded error 3	2	6	100
MS + seeded error 4	2	6	100
EL1 mutant	59	20	93
EL2 mutant	51	21	100

Saturation-based testing [Sherman, Dwyer, Elbaum FSE ‘09]

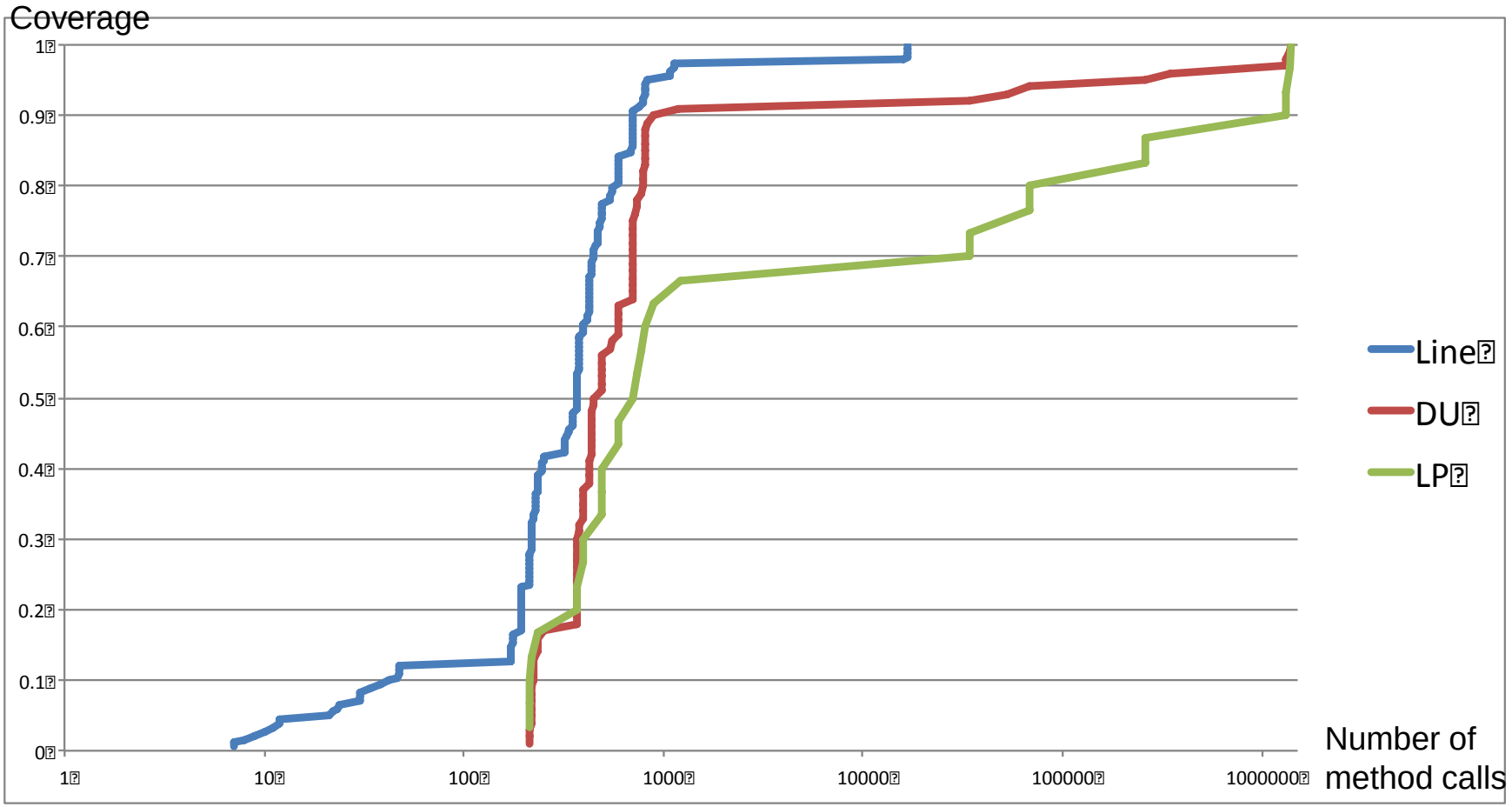
- Randomization, controlled exploration of thread interleavings
  - Do they yield coverage of behaviors related to concurrency-specific faults?
- “Adequate testing of concurrent programs requires stronger coverage metrics.”

- Coverage metrics must avoid being
  - too weak: saturate prematurely
  - too strong: hard to compute coverage target

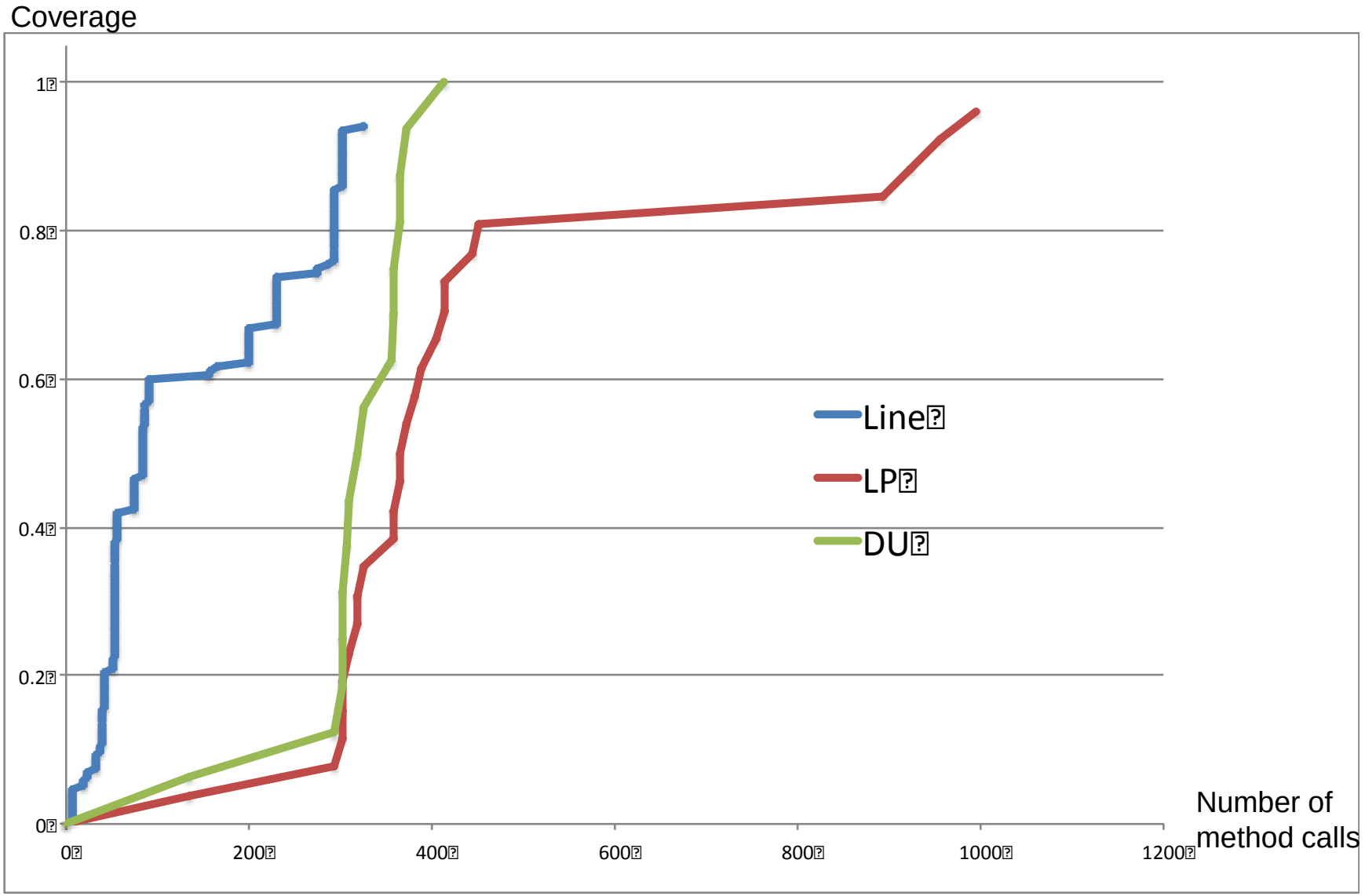
- Coverage denominator too hard to compute for non-trivial metrics
  - Use saturation to stop testing using a particular approach

- Our saturation experiments: LP is stronger than MP and DU
  - Not too strong: Saturates later but still within reasonable time

Saturation: Elevator



Saturation: SOR



Summary

- A coverage metric for shared-memory concurrent programs: location pairs

- Corresponds well to atomicity and refinement violations
- Better than all definitions-uses and method pairs

- More demanding than other metrics for concurrent programs
- Saturates later, but still within reasonable time

- Compromise between
  - difficulty of computing and attaining coverage target, and
  - bug detection ability