Effective Static Race Detection for Java

Mayur Naik

Alex Aiken

Stanford University

What is a Race?

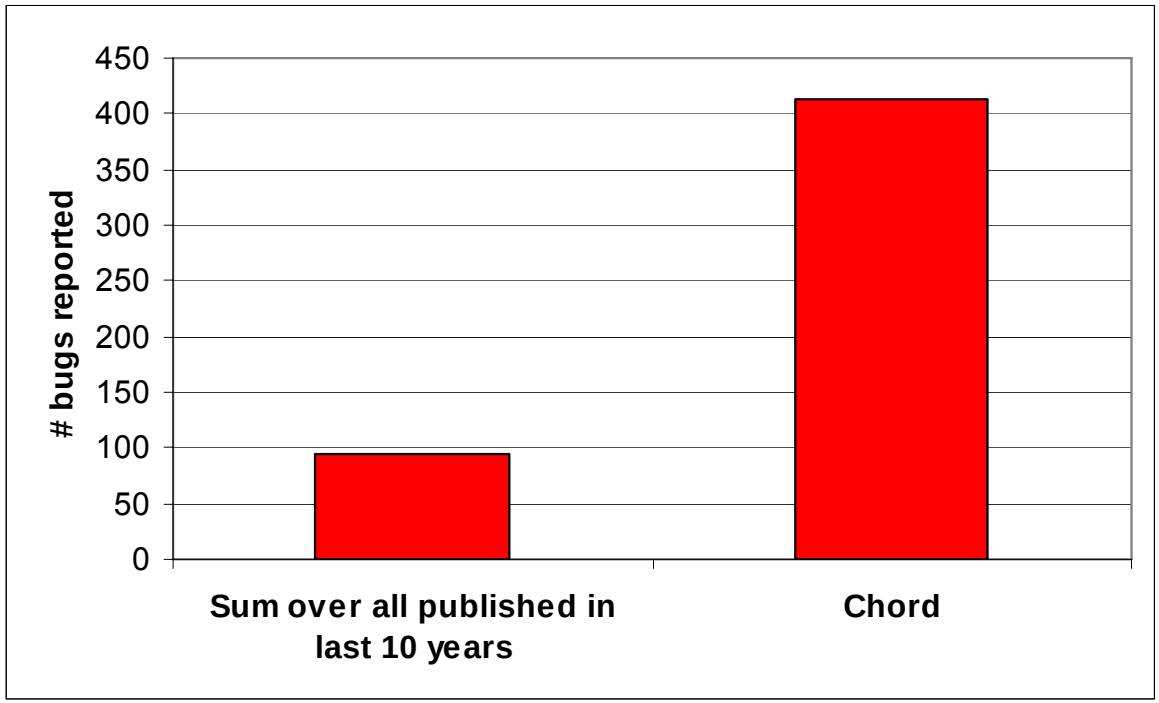
A condition in a shared-memory, multi-threaded program in which:

- same memory location accessed by different threads simultaneously (without holding a common lock)
- at least one of the accesses is a write

Why is Race Detection Important?

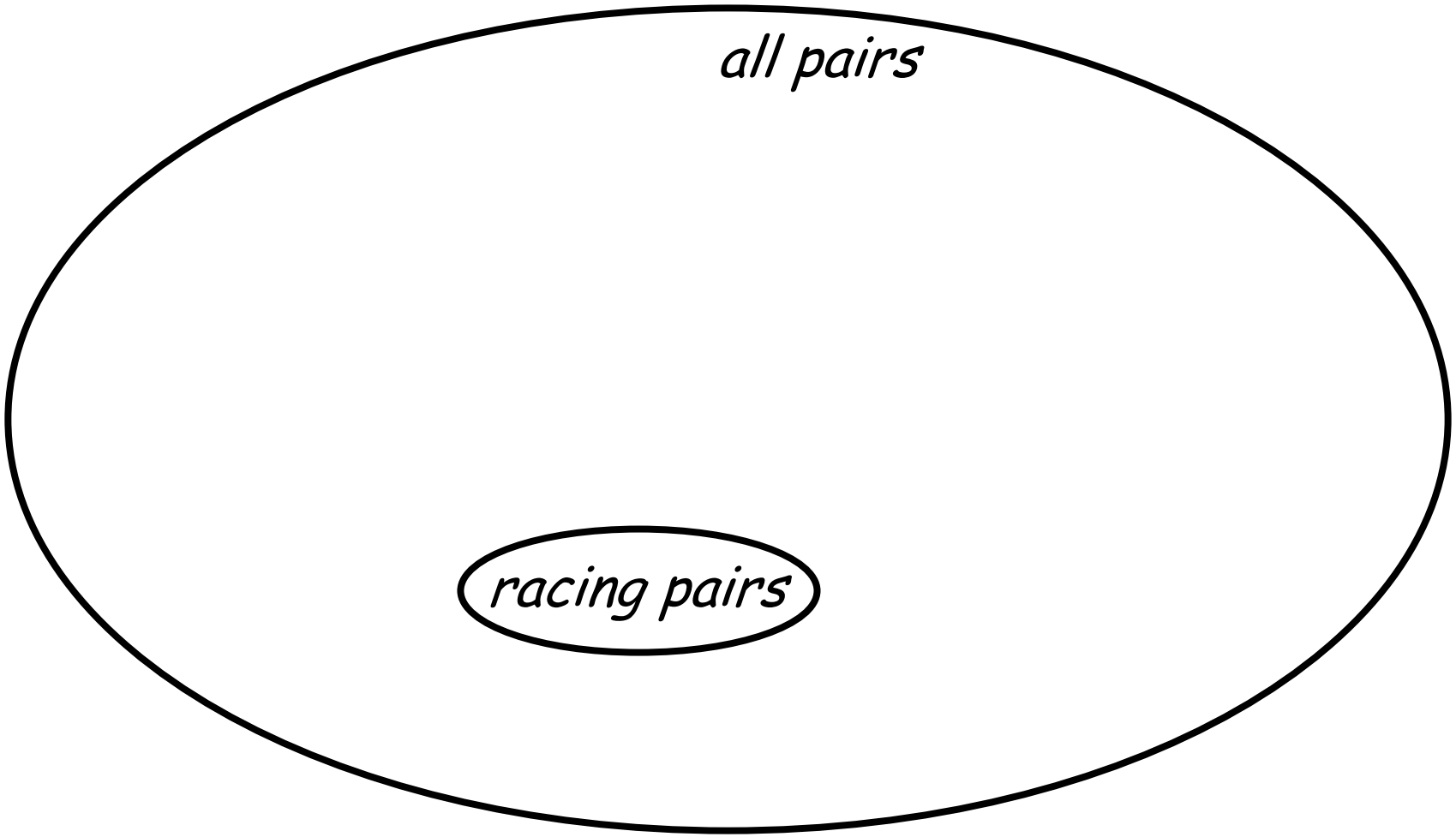
- Particularly insidious concurrency bug
  - Triggered non-deterministically
  - No fail-fast behavior even in safe languages like Java
- Fundamental in concurrency theory and practice
  - Lies at heart of many concurrency problems
    - atomicity checking, deadlock detection, ...
  - Today's concurrent programs riddled with races
    - "… most Java programs are so rife with concurrency bugs that they work only 'by accident'."
      – *Java Concurrency in Practice*, Addison-Wesley, 2006

Our Result



- 412 bugs in mature Java programs comprising 1.5 MLOC
  - Many fixed within a week by developers

Our Race Detection Approach

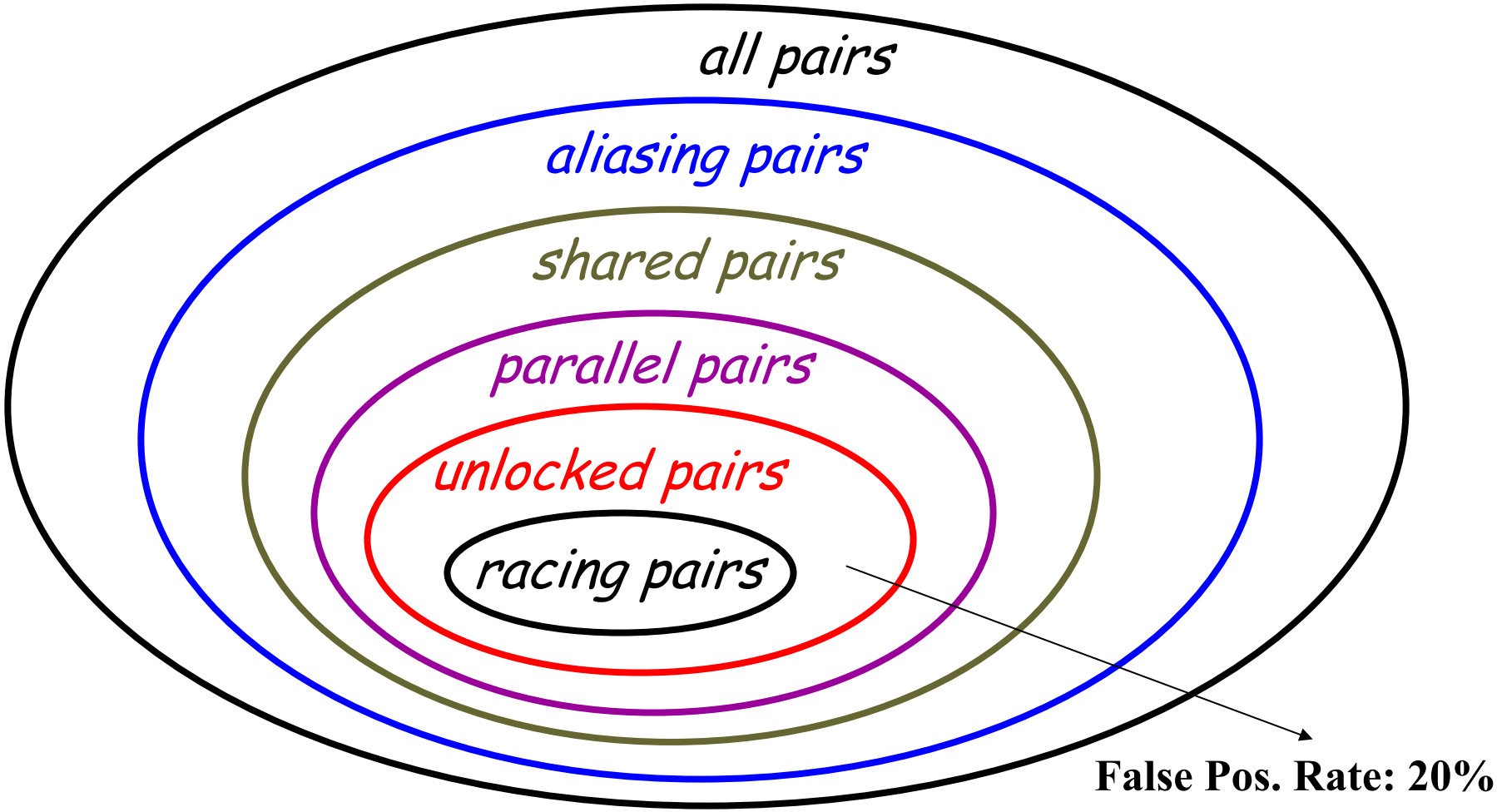


Challenges

Same location accessed by different threads simultaneously without common lock held

- Handle multiple aspects
  - Same location accessed
  - … by different threads
  - … simultaneously
- Correlate locks with locations they guard
  - … without common lock held
- Precision
  - Showed precise ***may alias analysis*** is central (PLDI'06)
  - low false-positive rate (20%)
- Soundness
  - Devised ***conditional must not alias analysis*** (POPL'07)
  - Circumvents ***must alias analysis***

Our Race Detection Approach



Same location accessed by different threads simultaneously without common lock held

Alias Analysis for Race Detection

```
// Thread 1:
sync (l1) {
    … e1.f …
}
```

```
// Thread 2:
sync (l2) {
    … e2.f …
}
```

- Field **f** is race-free if:

e1 and **e2** *never* refer to the same value

MUST-NOT-ALIAS(**e1**, **e2**)

k-Object-Sensitive May-Alias Analysis

- Recent body of work [Milanova et al. ISSTA'03]
- Problem: Context-insensitive analysis too imprecise!
- Solution: Context-sensitive analysis (k-CFA)
  - Abstract value = set of allocation sites
  - Context (k=1) = allocation site of `this` parameter

```
foo() {
    … e1.f …
}
```

```
bar() {
    … e2.f …
}
```

¬ **MAY-ALIAS(e1, e2)** if Sites(**e1**) ∩ Sites(**e2**) = ∅

```
foo() {
    e1.baz();
}
```

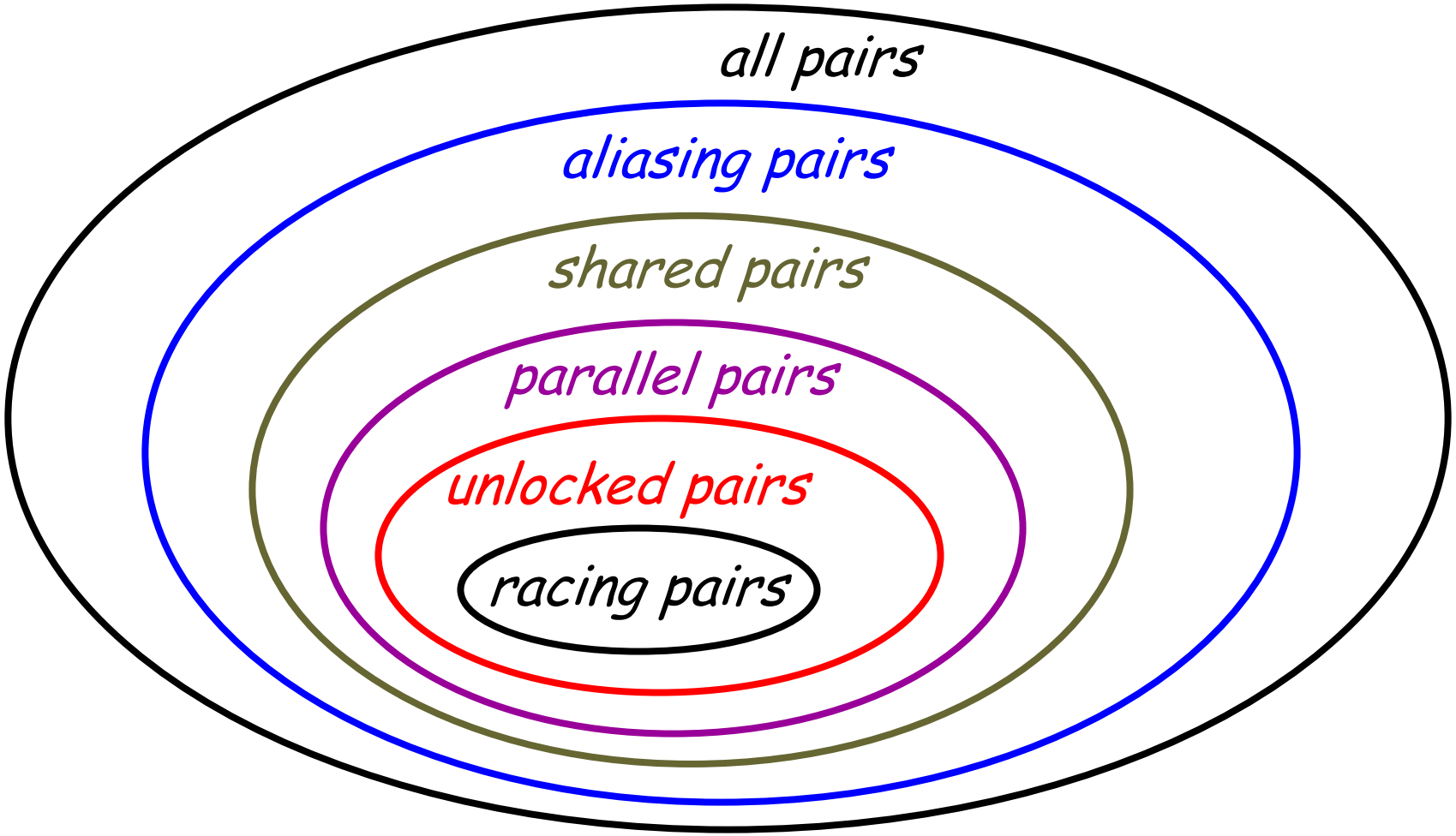
```
bar() {
    e2.baz();
}
```

Analyze function **baz** in two contexts

k-Object-Sensitive Analysis: Our Contributions

- No scalable implementations for even $k = 1$
- Insights:
  - Symbolic representation of relations
    - BDDs [Whaley-Lam PLDI'04, Lhotak-Hendren PLDI'04]
  - Demand-driven race detection algorithm
    - Begin with $k = 1$ for all allocation sites
    - Increment k only for those involved in races
- Allow scalability to $k = 7$

Our Race Detection Approach



Same location accessed by different threads simultaneously without common lock held

Alias Analysis for Race Detection

```
// Thread 1:
sync (l1) {
    … e1.f …
}
```

```
// Thread 2:
sync (l2) {
    … e2.f …
}
```

- Field **f** is race-free if:

e1 and **e2** *never* refer to the same value

¬ MAY-ALIAS(e1, e2)

OR

l1 and **l2** *always* refer to the same value

MUST-ALIAS(l1, l2)

Must Alias Analysis

- Small body of work
  - Much harder problem than may alias analysis
- Impediment to many previous race detection approaches
  - Folk wisdom: Static race detection is intractable

Insight: Must alias analysis not necessary for race detection!

New Idea: Conditional Must Not Alias Analysis

```
// Thread 1:
sync (l1) {
    … e1.f …
}
```

```
// Thread 2:
sync (l2) {
    … e2.f …
}
```

- Field **f** is race-free if:

  Whenever **l1** and **l2** refer to different values, **e1** and **e2** also refer to different values

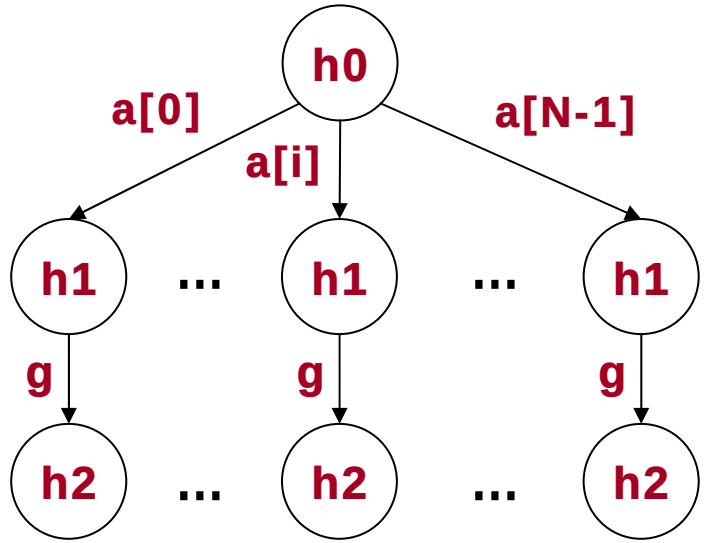
  MUST-NOT-ALIAS(l1, l2) => MUST-NOT-ALIAS(e1, e2)

Example

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```

```
x1 = a[*];
sync (?) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (?) {
    x2.g.f = …;
}
```

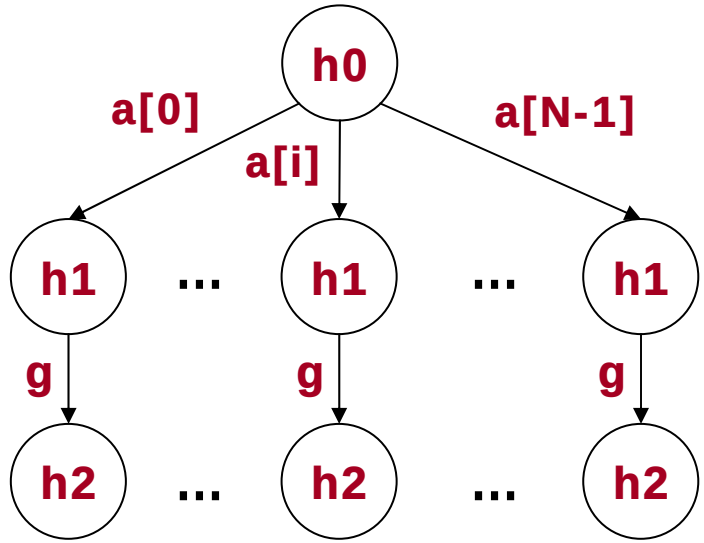


Easy Case: Coarse-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```

```
x1 = a[*];
sync (a) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (a) {
    x2.g.f = …;
}
```



Field `f` is race-free if:

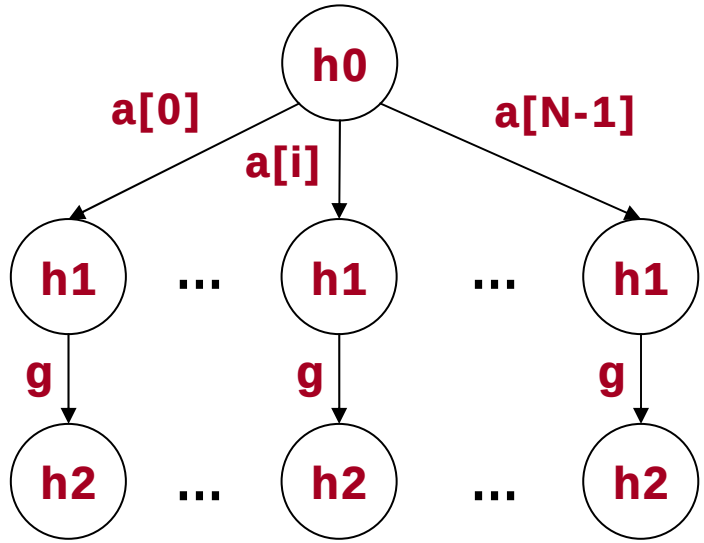
MUST-NOT-ALIAS(a1, a2) ⇒ MUST-NOT-ALIAS(x1.g, x2.g)

Example

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```

```
x1 = a[*];
sync (?) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (?) {
    x2.g.f = …;
}
```

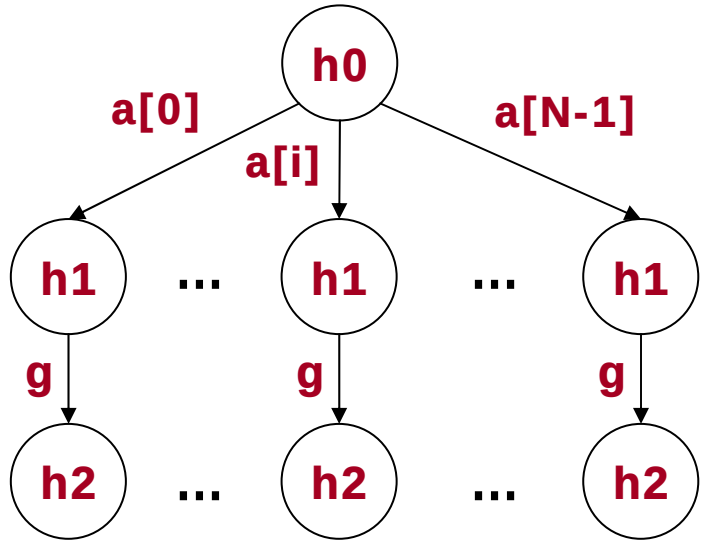


Easy Case: Fine-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```

```
x1 = a[*];
sync (x1.g) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (x2.g) {
    x2.g.f = …;
}
```



Field `f` is race-free if:

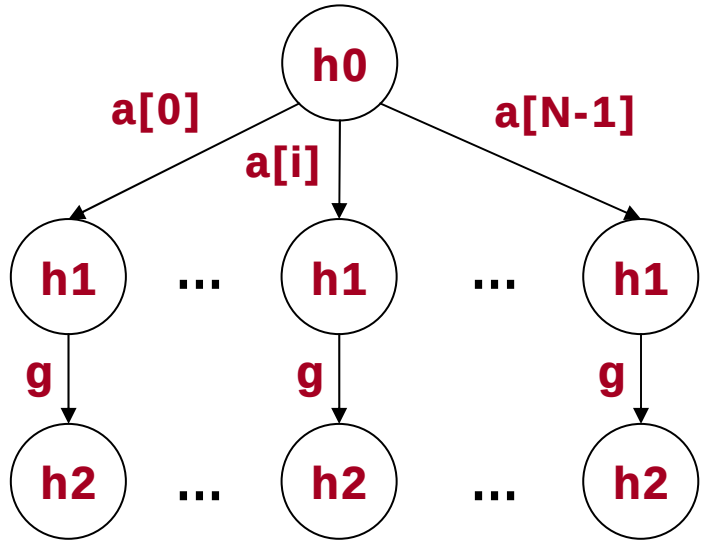
MUST-NOT-ALIAS(x1, x2) ⇒ MUST-NOT-ALIAS(x1.g, x2.g)

Example

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```

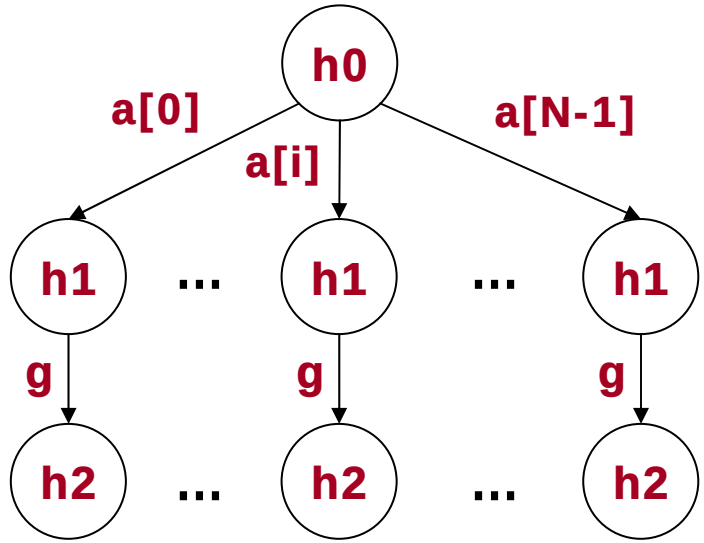
```
x1 = a[*];
sync (?) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (?) {
    x2.g.f = …;
}
```



Hard Case: Medium-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```




```
x1 = a[*];              x2 = a[*];
sync (x1) {             sync (x2) {
    x1.g.f = …;             x2.g.f = …;
}                       }
```

Field `f` is race-free if:

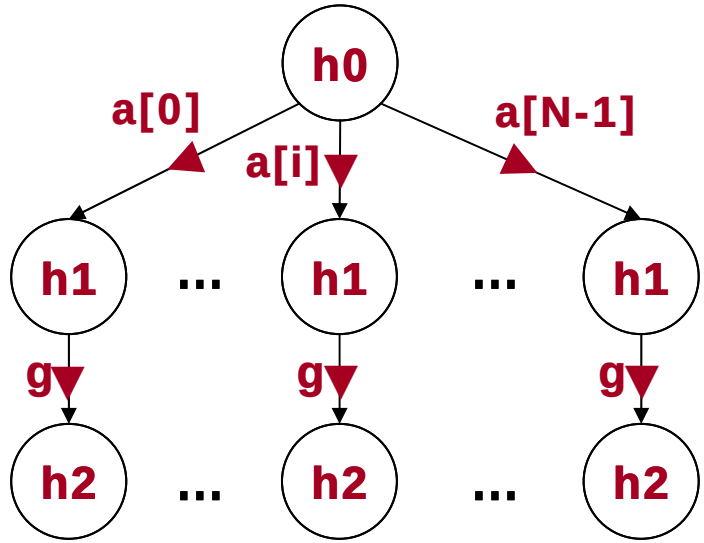
`MUST-NOT-ALIAS(x1, x2)` ⇒ `MUST-NOT-ALIAS(x1.g, x2.g)`

(field of distinct h1 values linked to distinct h2 values)

Disjoint Reachability

In every execution, if:

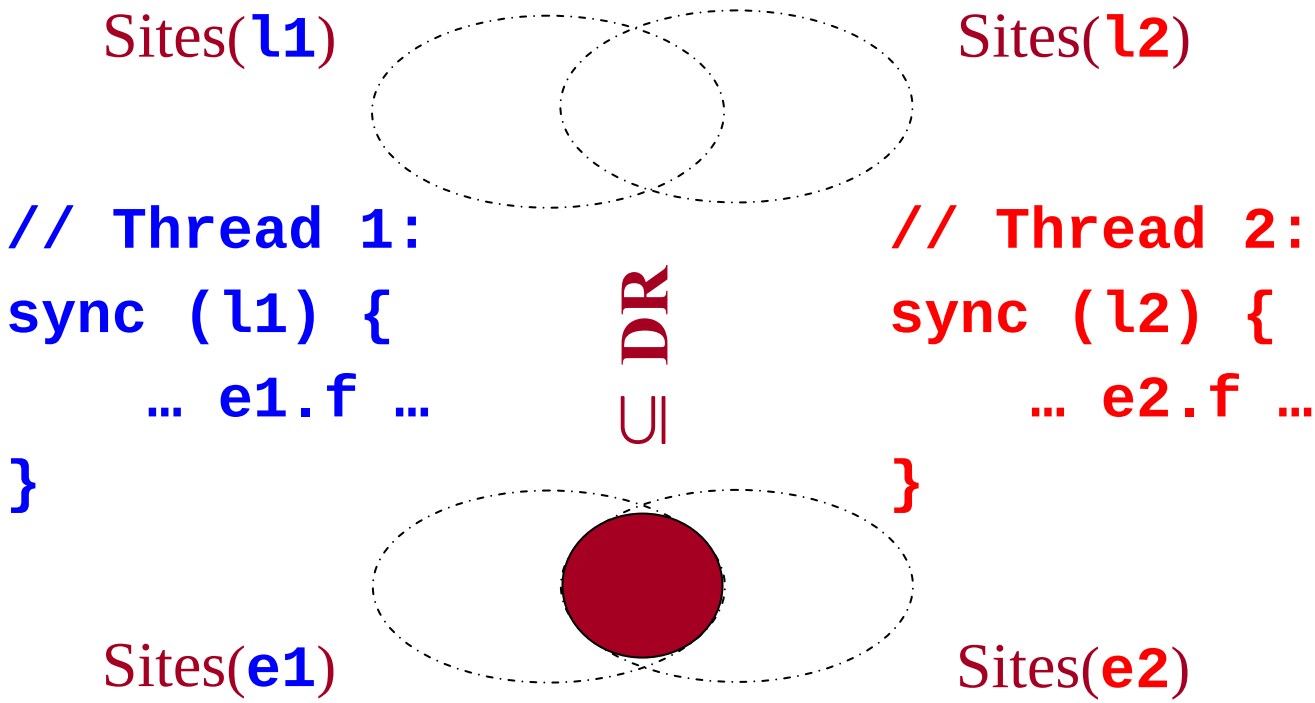
- from *distinct* **h1** values
- we can reach (via 1 or more fields)
- only *distinct* **h2** values



then **{h2} ⊆ DR({h1})**

Note: Values abstracted by sets of allocation sites

Conditional Must Not Alias Analysis using Disjoint Reachability

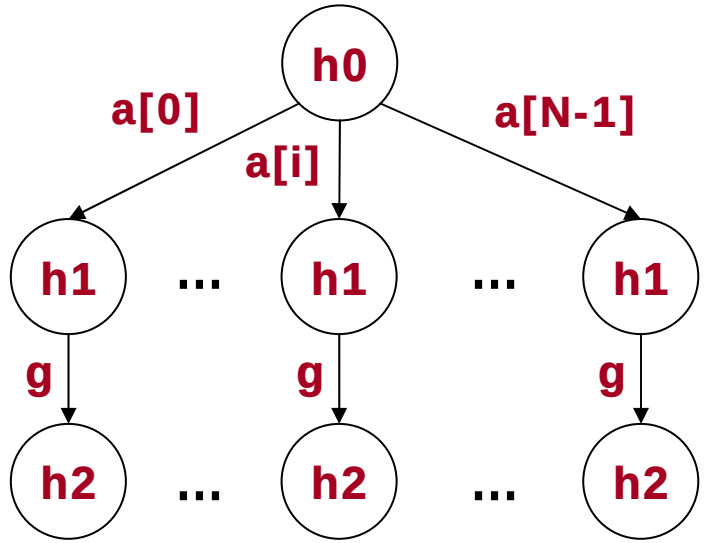


Field **f** is race-free if:

- $(Sites(e1) \cap Sites(e2)) \subseteq DR(Sites(l1) \cup Sites(l2))$ ⇒ MUST-NOT-ALIAS(**e1**, **e2**)
- MUST-NOT-ALIAS(**l1**, **l2**)
  - **e1** reachable from **l1** and **e2** reachable from **l2**

Hard Case: Medium-grained Locking

```
a = new h0[N];
for (i = 0; i < N; i++) {
    a[i] = new h1;
    a[i].g = new h2;
}
```




```
x1 = a[*];
sync (x1) {
    x1.g.f = …;
}
```

```
x2 = a[*];
sync (x2) {
    x2.g.f = …;
}
```

Field **f** is race-free if:

- (pairs(x1) ∩ pairs(x2) = ∅) ∨ (DRSides(x1) ∩ DRSides(x2) = ∅)
- true reachable from x1 and x2 reachable from x2

Experience with Chord

- Experimented with 12 multi-threaded Java programs
  - smaller programs used in previous work
  - larger, mature and widely-used open-source programs
  - whole programs and libraries

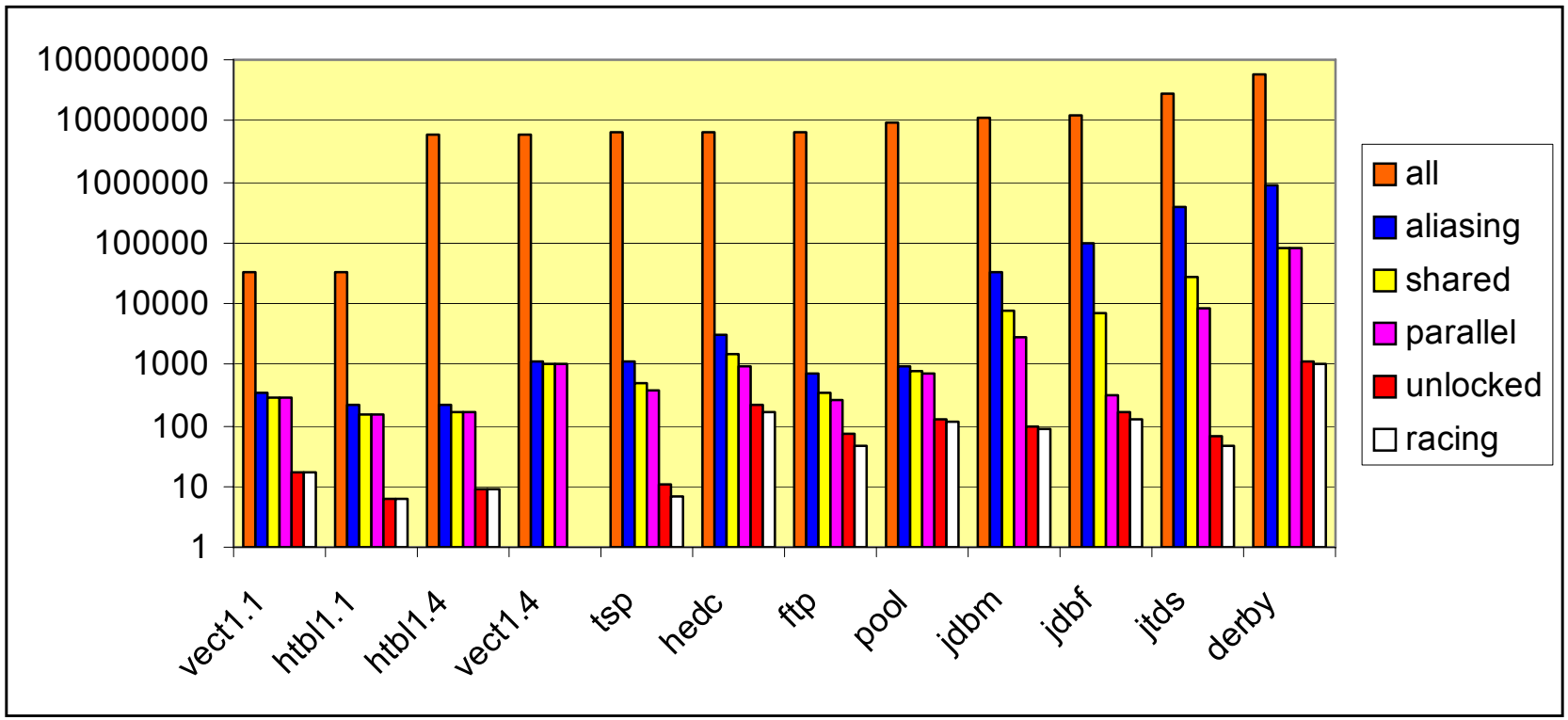
- Tool output and developer discussions available at: http://www.cs.stanford.edu/~mhn/chord.html

- Programs being used by other researchers in race detection

Benchmarks

	classes	KLOC	description	time
vect1.1	**19**	**3**	JDK 1.1 java.util.Vector	**0m28s**
htbl1.1	**21**	**3**	JDK 1.1 java.util.Hashtable	**0m27s**
htbl1.4	**366**	**75**	JDK 1.4 java.util.Hashtable	**2m04s**
vect1.4	**370**	**76**	JDK 1.4 java.util.Vector	**2m02s**
tsp	**370**	**76**	Traveling Salesman Problem	**3m03s**
hedc	**422**	**83**	Web crawler	**9m10s**
ftp	**493**	**103**	Apache FTP server	**11m17s**
pool	**388**	**124**	Apache object pooling library	**10m29s**
jdbm	**461**	**115**	Transaction manager	**9m33s**
jdbf	**465**	**122**	O/R mapping system	**9m42s**
jtds	**553**	**165**	JDBC driver	**10m23s**
derby	**1746**	**646**	Apache RDBMS	**36m03s**

Pairs Retained After Each Stage (Log scale)



Classification of Unlocked Pairs

	harmful	benign	false	# bugs
vect1.1	**5**	**12**	**0**	**1**
htbl1.1	**0**	**6**	**0**	**0**
htbl1.4	**0**	**9**	**0**	**0**
vect1.4	**0**	**0**	**0**	**0**
tsp	**7**	**0**	**4**	**1**
hedc	**170**	**0**	**41**	**6**
ftp	**212**	**3**	**43**	**32**
pool	**105**	**10**	**13**	**17**
jdbm	**91**	**0**	**7**	**2**
jdbf	**130**	**0**	**34**	**18**
jtds	**34**	**14**	**17**	**16**
derby	**1018**	**0**	**78**	**319**

Developer Feedback

- 16 bugs in jTDS
  - Before: “As far as we know, there are no concurrency issues in jTDS …”
  - After: “It is probably the case that the whole synchronization approach in jTDS should be revised from scratch ...”
- 17 bugs in Apache Commons Pool
  - “Thanks to an audit by Mayur Naik many potential synchronization issues have been fixed” -- *Release notes for Commons Pool 1.3*
- 319 bugs in Apache Derby
  - “This looks like *very* valuable information and I for one appreciate you using Derby … Could this tool be run on a regular basis? It is likely that new races could get introduced as new code is submitted ...”

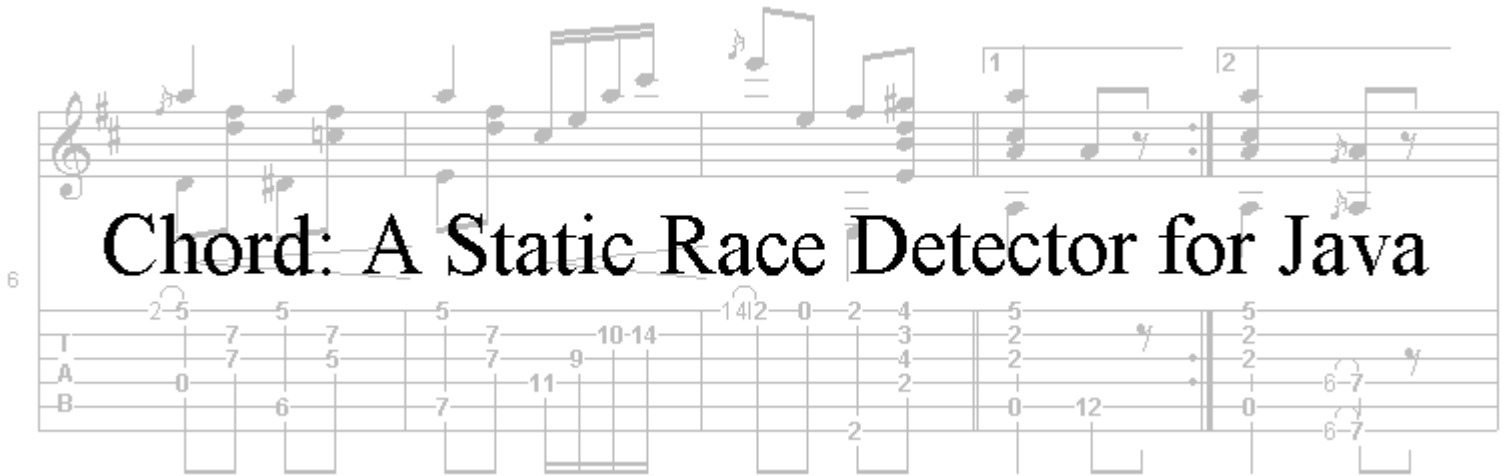
Related Work

- Static (compile-time) race detection
  - Need to approximate multiple aspects
  - Need to perform must alias analysis
  - Sacrifice precision, soundness, scalability
- Dynamic (run-time) race detection
  - Current state of the art
  - Inherently unsound
  - Cannot analyze libraries
- Shape Analysis
  - much more expensive than disjoint reachability

Summary of Contributions

- Precise race detection (PLDI'06)
  - Key idea: *k*-object-sensitive may alias analysis
  - Important client for may alias analyses
- Sound race detection (POPL'07)
  - Key idea: Conditional must not alias analysis
  - Has applications besides race detection
- Effective race detection
  - 412 bugs in mature Java programs comprising 1.5 MLOC
  - Many fixed within a week by developers

The End



http://www.cs.stanford.edu/~mhn/chord.html