D4:
Fast Concurrency Debugging with Parallel Differential Analysis

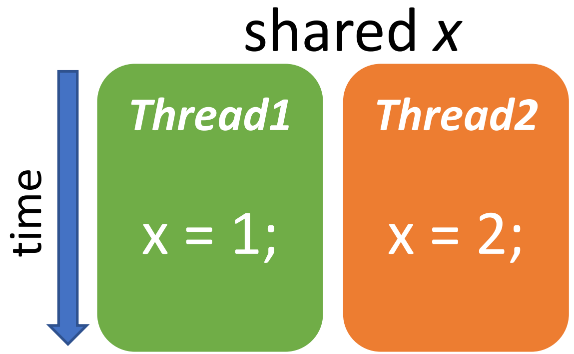
Bozhen Liu **Jeff Huang**

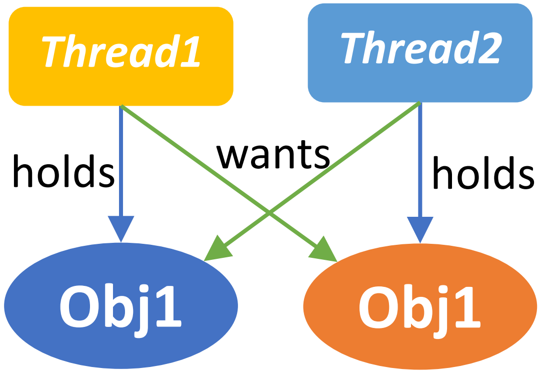
Parasol Lab, Texas A&M University

TEXAS A&M
UNIVERSITY

Concurrency Bugs:

- easy to introduce
- hard to detect and debug
- data races
- deadlocks





Existing Solutions

RacerX [SOSP'03]
Chord [PLDI'06]
FastTrack [PLDI'09]
RVPredict [PLDI'14]
Vindicator [PLDI'18]
…

TOO LATE!

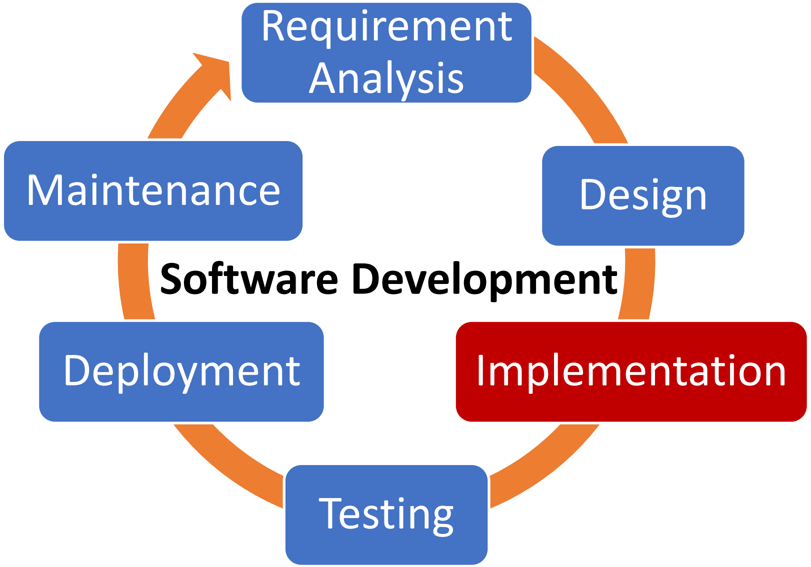
Other Solutions?

Nip the bugs in the bud?

Our prior work
Zhan and Huang [FSE'16]

YES! **ECHO**[1]: Incremental detection in the programming phase!

- 92% ⇒ < 0.1s
- Instant feedback

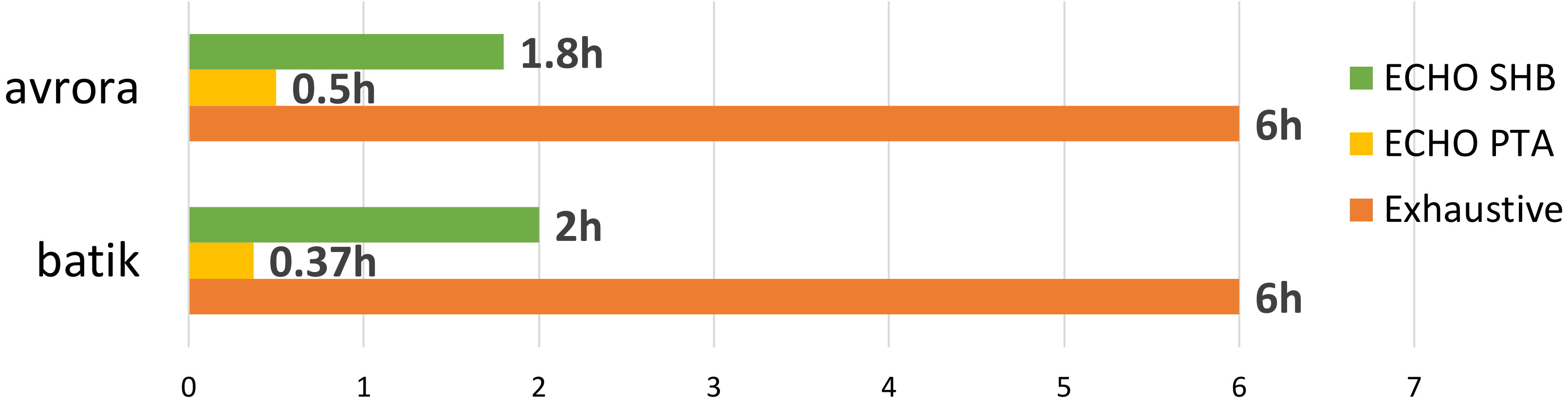


Other Solutions?

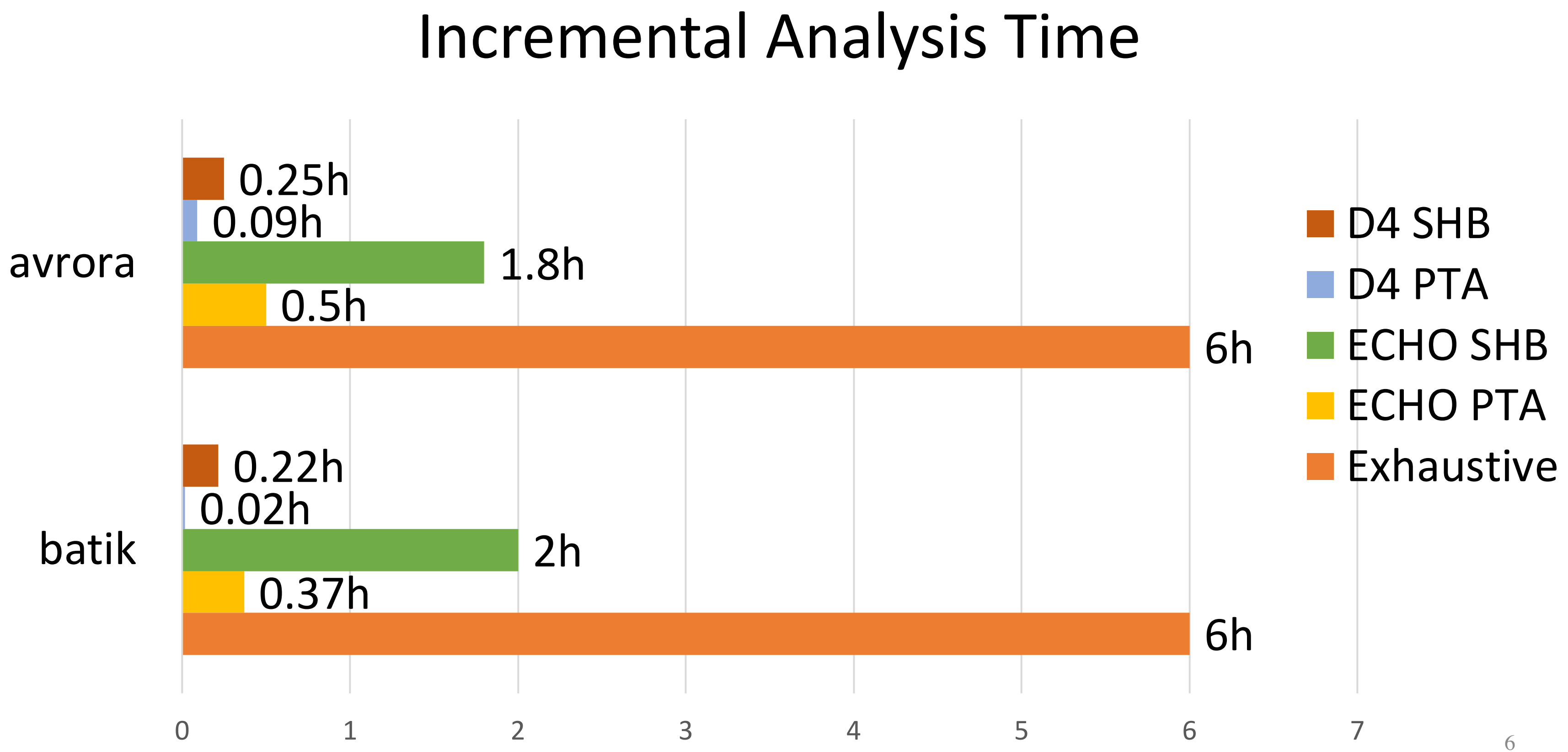
Instant feedback for large programs?

NO!   Slow in large programs!

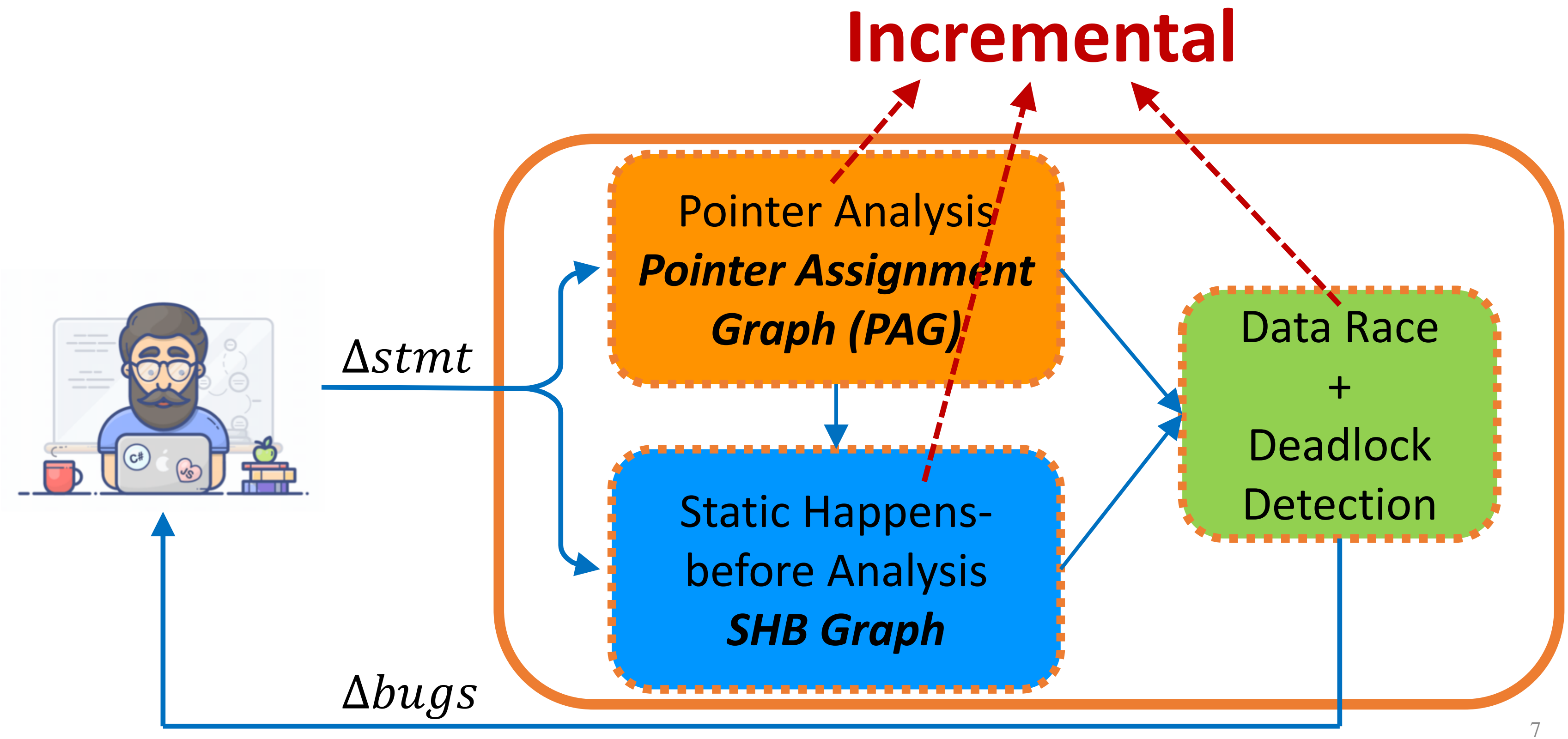
Incremental Analysis Time



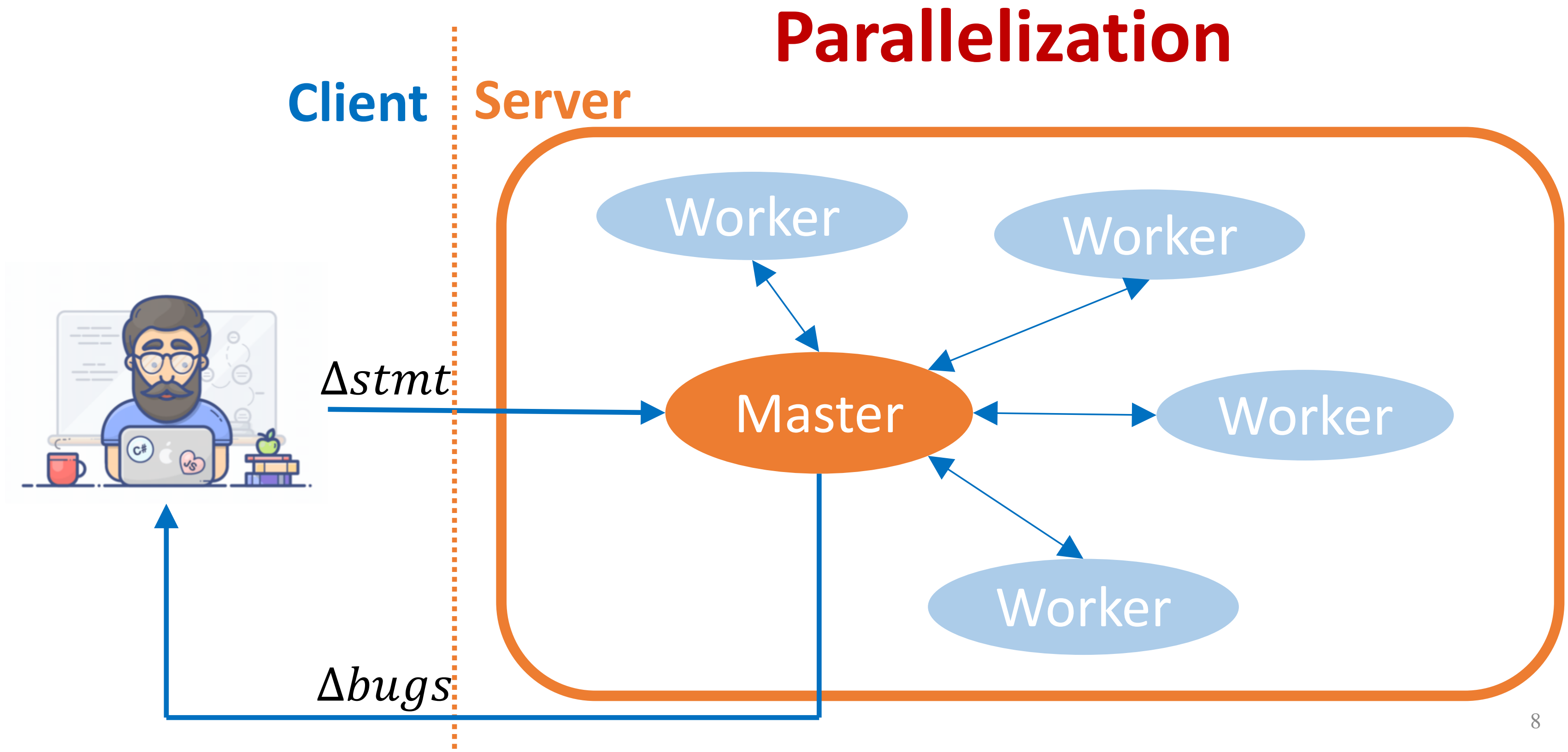
D4


Incremental Analysis Time
avrora
batik
0.25h
0.09h
1.8h
0.5h
6h
0.22h
0.02h
2h
0.37h
6h
0
1
2
3
4
5
6
7
D4 SHB
D4 PTA
ECHO SHB
ECHO PTA
Exhaustive

Our Solution to Scalability



Our Solution to Scalability

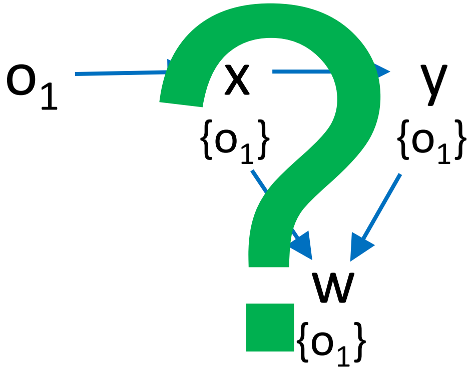


Incremental Pointer Analysis

- Addition ⇒ Easy
- Deletion ⇒ Hard
- Modification

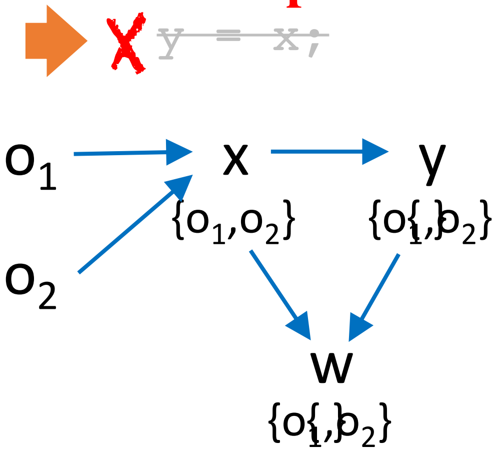
  = Delete Old Stmt + Add New Stmt

```
  x = new C();//o1
✗ y = x;
  w = y;
  x = w;
+ x = new C();//o2
```

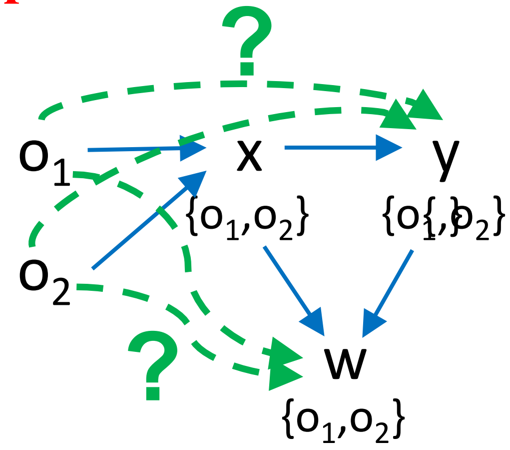


Two Existing Incremental Algorithms To Handle Deletion

- Reset-Recompute Algorithm
  - Reset
  - Recompute
- **Redundant computation**



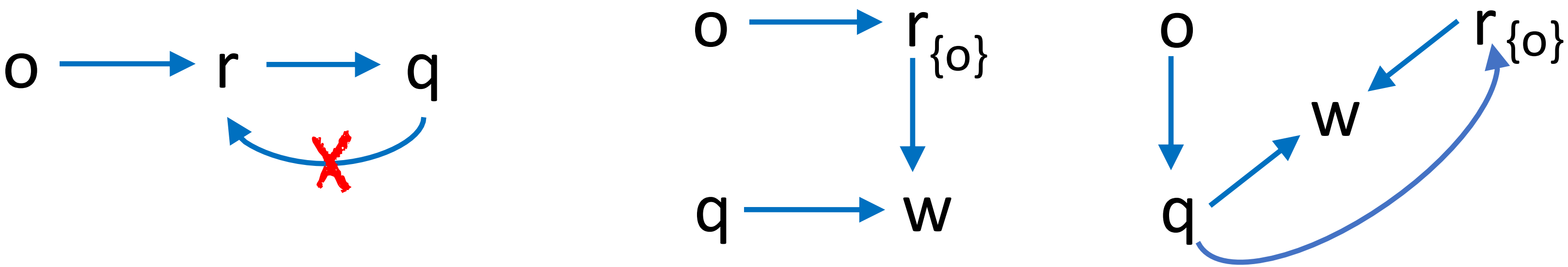
- Reachability-based Algorithm
  - Remove Edge
  - Check reachability
- **Expensive**



Our New Incremental Pointer Analysis

Key Insight: Local neighboring properties

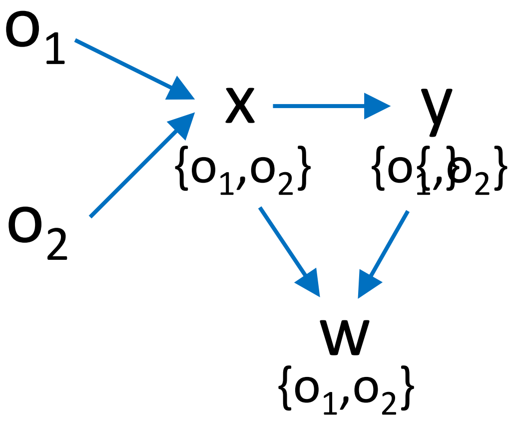
To incrementally update the PAG, it is sufficient to check the local neighbors of the nodes corresponding to the changed statement



Example

✗ ~~y = x;~~

1. Remove the edge
2. y
3. Check the incoming neighbours of y
4. Update pts(y)
5. Propagate $\{o_1, o_2\}$ to outgoing neighbours of y



Example

X ~~`y = x;`~~

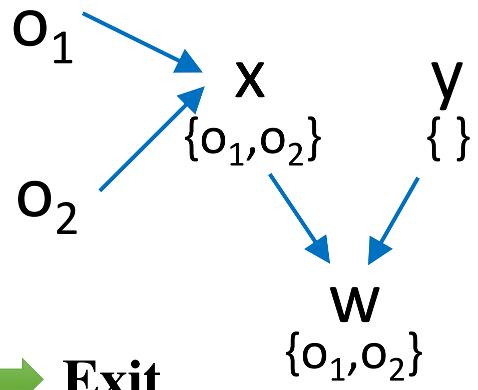
1.Remove the edge

2.w

3.Check the incoming neighbours of w ⟶ **Exit**

4.Update pts(w)

5.Propagate $\{o_1, o_2\}$ to outgoing neighbours of w



Our New Incremental Pointer Analysis

Multiple edge changes?

- Solve only one edge in each fix-point iteration

Cycles in a PAG?

- SCC[1]
- Incremental SCC detection

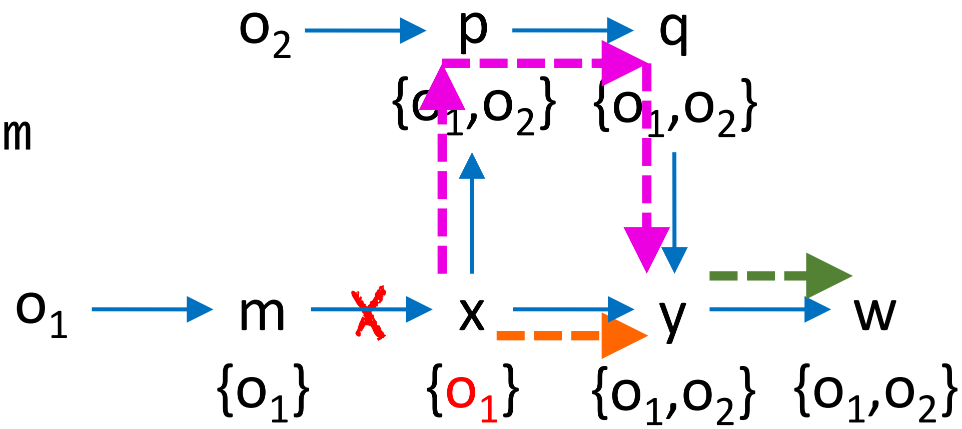
[1] HARDEKOPF, B., AND LIN, C. *The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code*. PLDI'07

Parallelization

Key Insight: Change consistency property

For an edge addition or deletion, if the change propagates to a node from multiple incoming neighbors, then the modification applied to the corresponding points-to set must be the same.

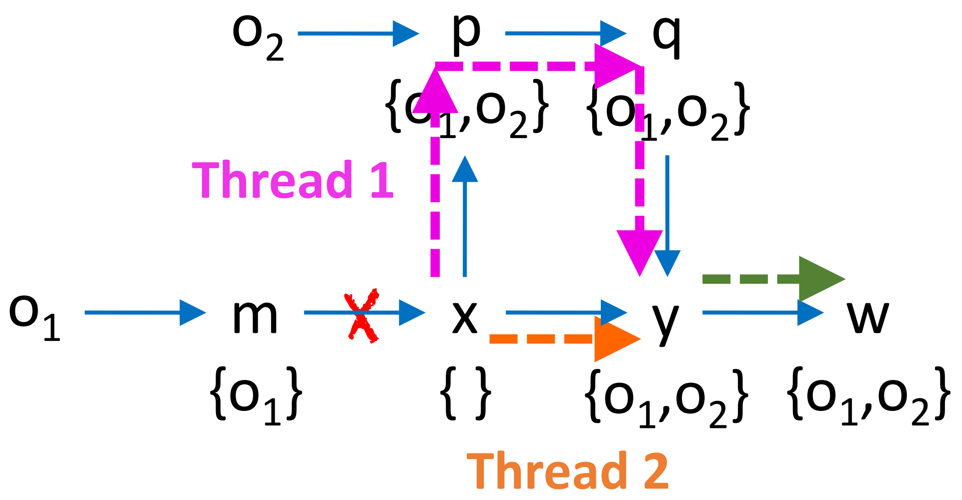
Delete x = m



Example

The change $\{o_1\}$ can be propagated to ***y*** first from

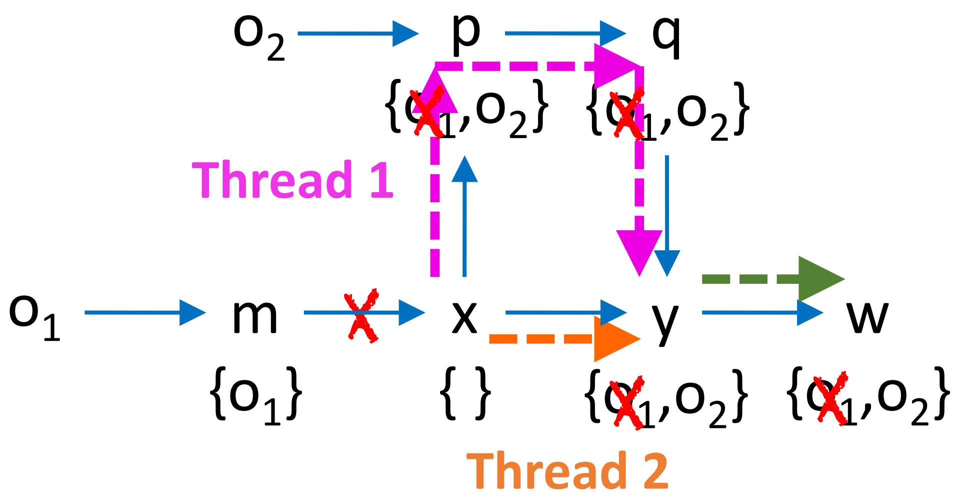
1) ***q***: the purple path
2) ***x***: the orange path



Example

The change $\{o_1\}$ can be propagated to ***y*** first from

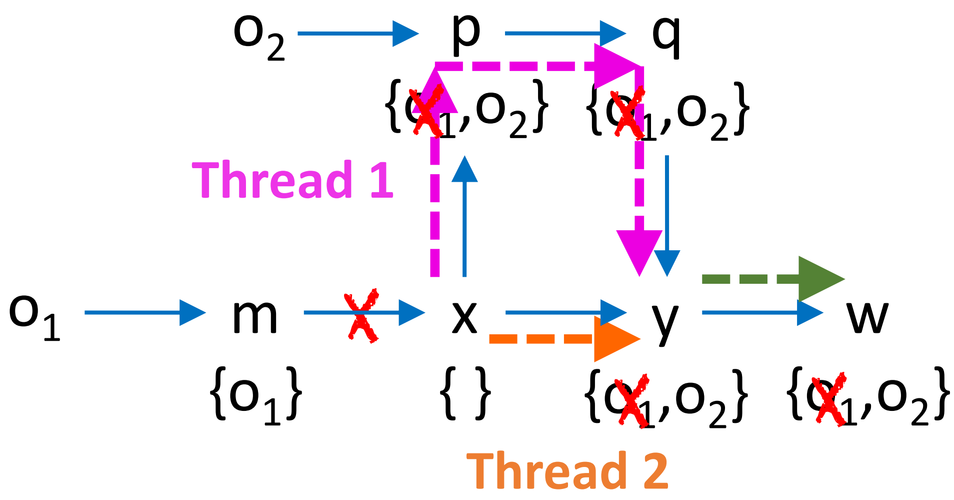
1) ***q***: the purple path
2) ***x***: the orange path



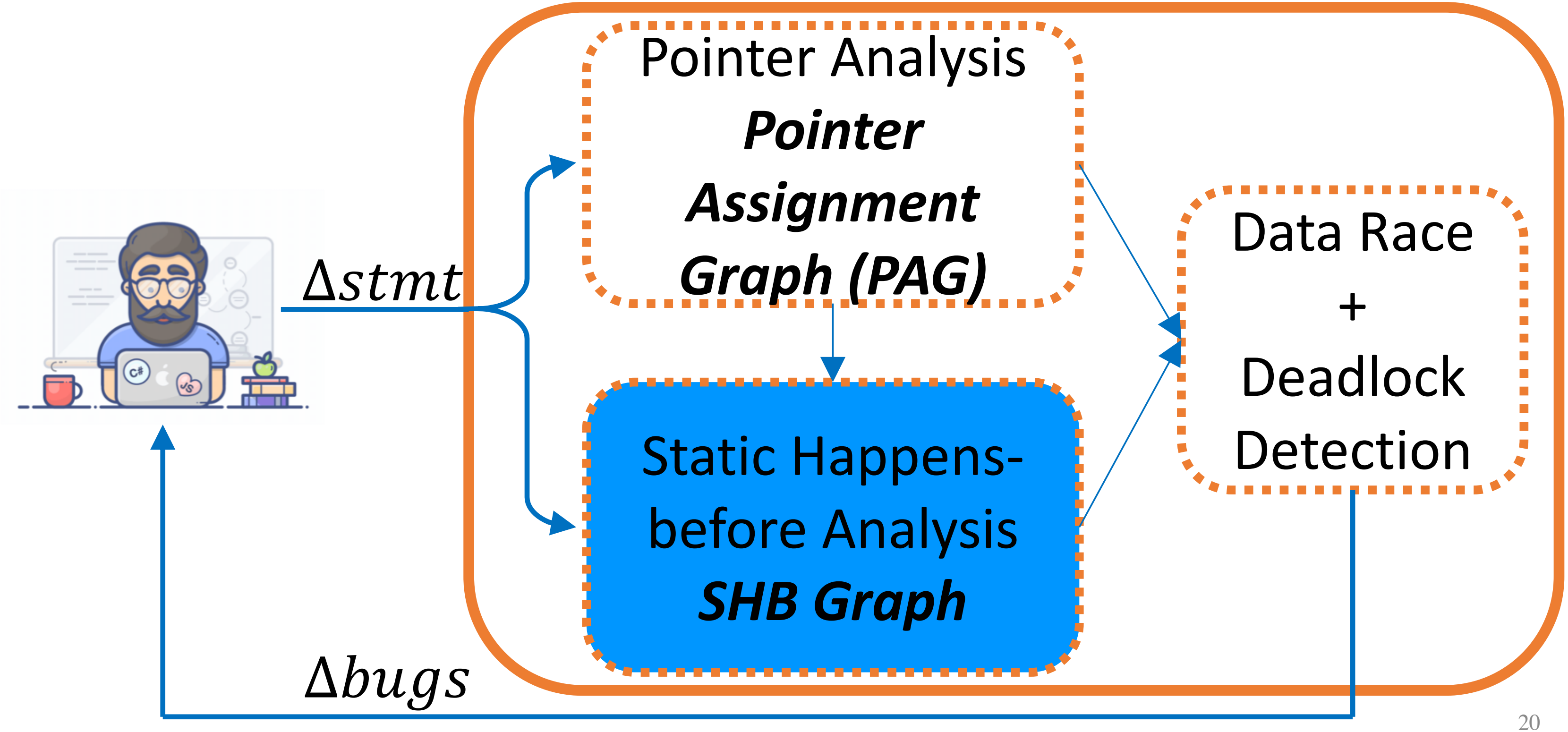
Example

The change {o_1} can be propagated to ***y*** first from

1) ***q***: the purple path
2) ***x***: the orange path



Overview: D4



Δstmt
Pointer Analysis
Pointer Assignment Graph (PAG)
Static Happens-before Analysis
SHB Graph
Data Race
+
Deadlock
Detection
Δbugs

Existing Static Happens-before Analysis

```
main(){
  x = 0;
  y = 5;
  t1 = new Thread();
  t2 = new Thread();
  t1.start();
  t2.start();
}

t1:
   x = 1;
   m1();
   m2();

t2:
   y = x;
   m1();
   m2();
```

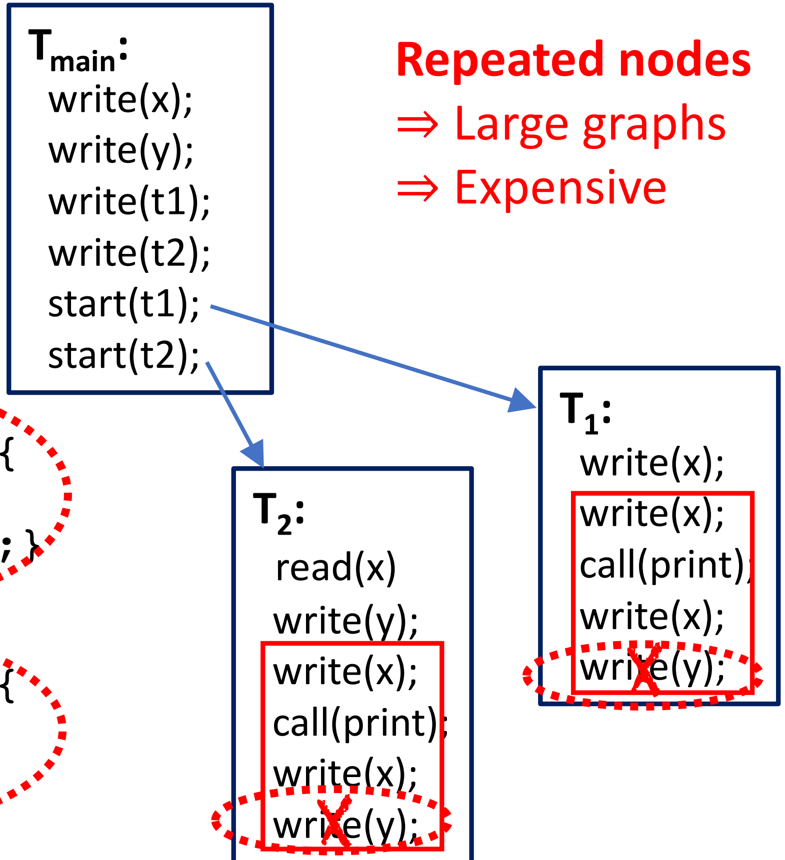
```
void m1(){
  x = 3;
  print(x);}

void m2(){
  x = 2;
  y = 0;}
```

Repeated nodes

⇒ Large graphs

⇒ Expensive



Our New Static Happens-before Analysis

Key Insight:* *A unique subgraph for each method/thread

SubSHB*: *Sub Static Happens-before Graph

- {thread id}
- Advantages:
  - No repeated nodes
  - A compact graph
  - Fast construction and incremental updates

Our New Static Happens-before Analysis

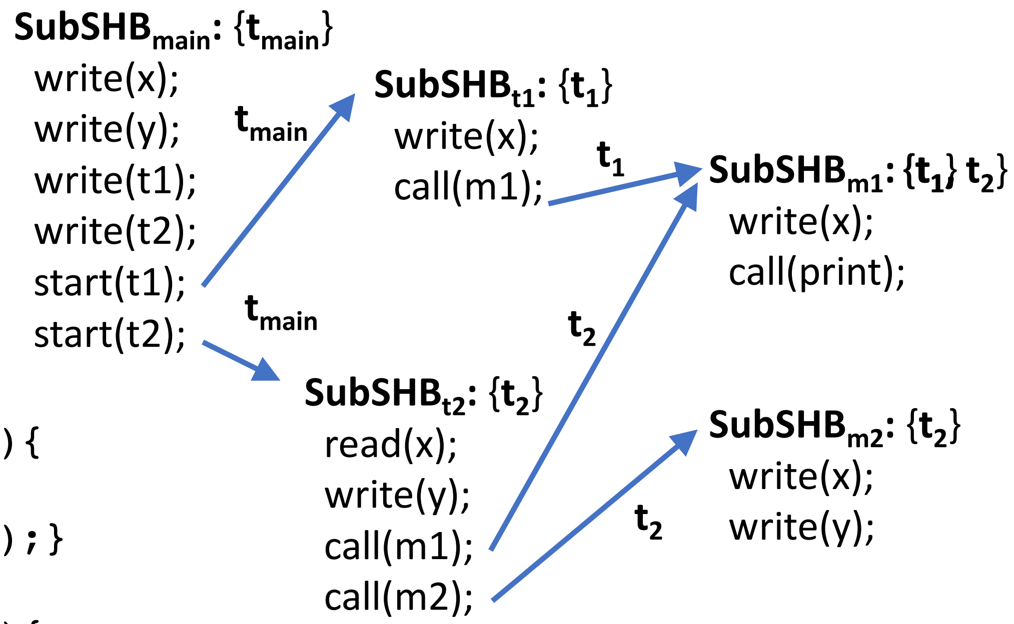
```
main() {
  x = 0;
  y = 5;
  t1 = new Thread();
  t2 = new Thread();
  t1.start();
  t2.start();
}
```

```
t1:
   x = 1;
   m1();
```

```
void m1(){
 x = 3;
 print(x);}
```

```
t2:
   y = x;
   m1();
   m2();
```

```
void m2(){
 x = 2;
 y = 0;}
```



No repeated nodes

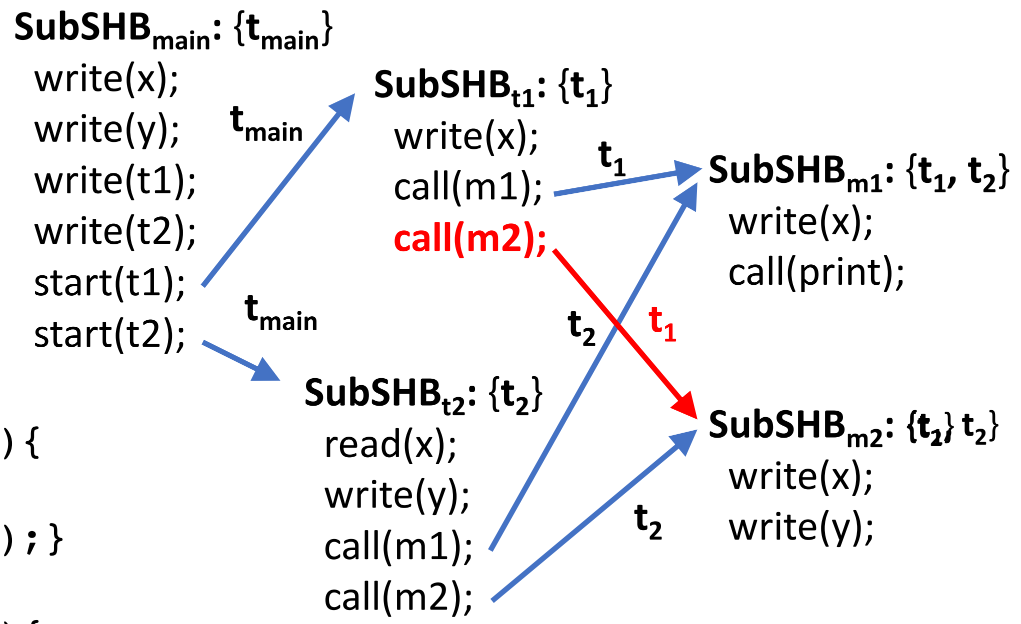
Our New Static Happens-before Analysis

```
main() {
  x = 0;
  y = 5;
  t1 = new Thread();
  t2 = new Thread();
  t1.start();
  t2.start();
}
```

```
t1:
   x = 1;
   m1();
   m2();
t2:
   y = x;
   m1();
   m2();
```

```
void m1(){
  x = 3;
  print(x);}

void m2(){
  x = 2;
  y = 0;}
```



Our New Static Happens-before Analysis

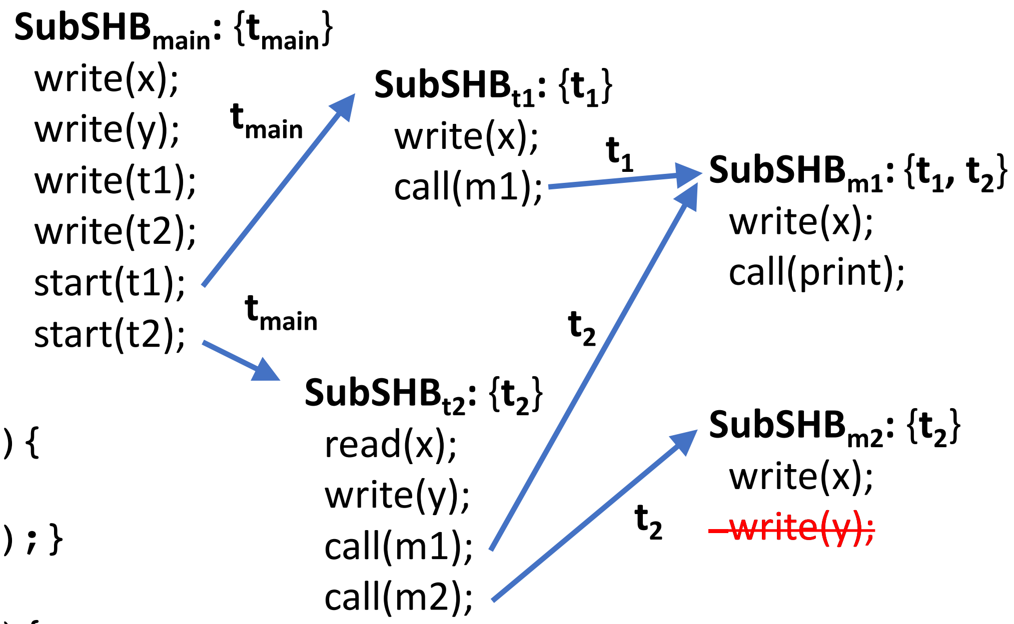
```
main() {
  x = 0;
  y = 5;
  t1 = new Thread();
  t2 = new Thread();
  t1.start();
  t2.start();
}
```

```
t1:
    x = 1;
    m1();

t2:
    y = x;
    m1();
    m2();
```

```
void m1(){
  x = 3;
  print(x);}

void m2(){
  x = 2;
  ~~y = 0;~~}
```



Efficient update ⇒ A node/an edge

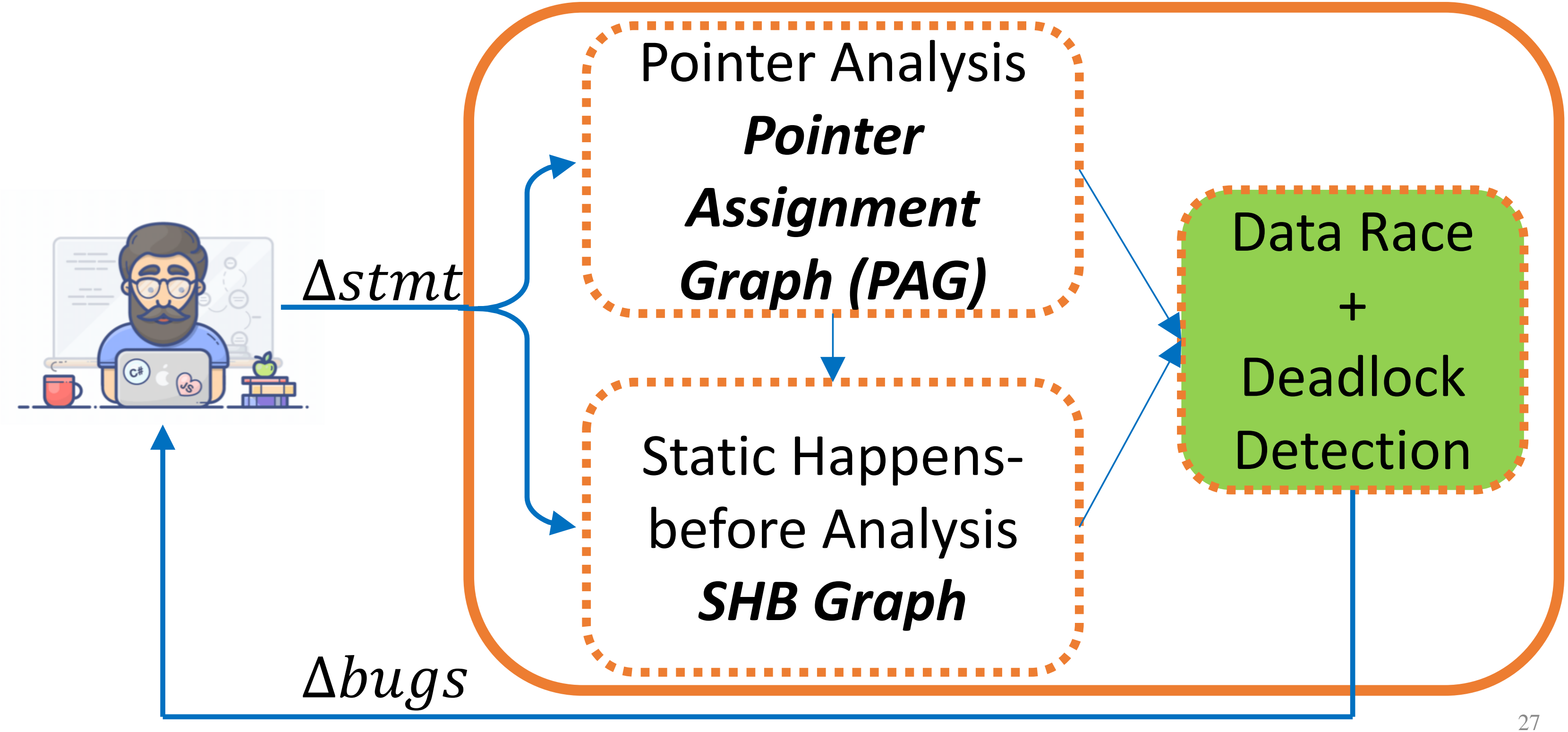
Parallelization

Key Insight:

Statements in different threads/methods are independent

- Different SubSHBs can be created in parallel
- Changes in different SubSHBs can be updated in parallel

Overview: D4


$\Delta stmt$
Pointer Analysis
Pointer Assignment Graph (PAG)
Static Happens-before Analysis
SHB Graph
Data Race + Deadlock Detection
$\Delta bugs$

Data Races

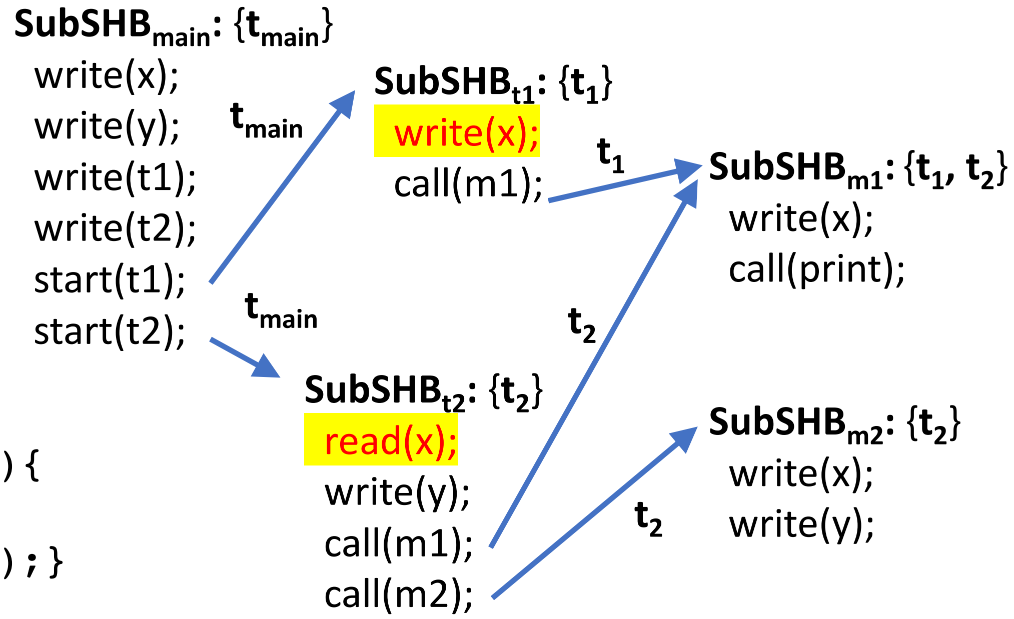
```
1 main() {
2   x = 0;
3   y = 5;
4   t1 = new Thread();
5   t2 = new Thread();
6   t1.start();
7   t2.start();
8 }
```

```
t1:
9    x = 1;
10   m1();

t2:
11   y = x;
12   m1();
13   m2();
```

```
14  void m1(){
15   x = 3;
16   print(x);}

17  void m2(){
18   x = 2;
19   y = 0;}
```



Race: x: line 9, line 11

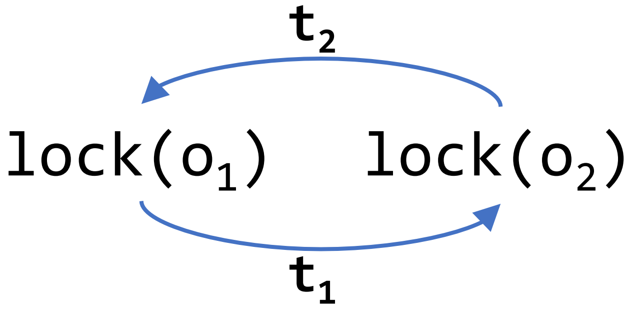
Deadlocks : Lock-dependency Graph

t₁:

```
synchronized(m){//pts(m)={o1}
  synchronized(n){//pts(n)={o2}
   ...
  }
}
```

t₂:

```
synchronized(p){//pts(p)={o2}
  synchronized(q){//pts(q)={o1}
   ...
  }
}
```



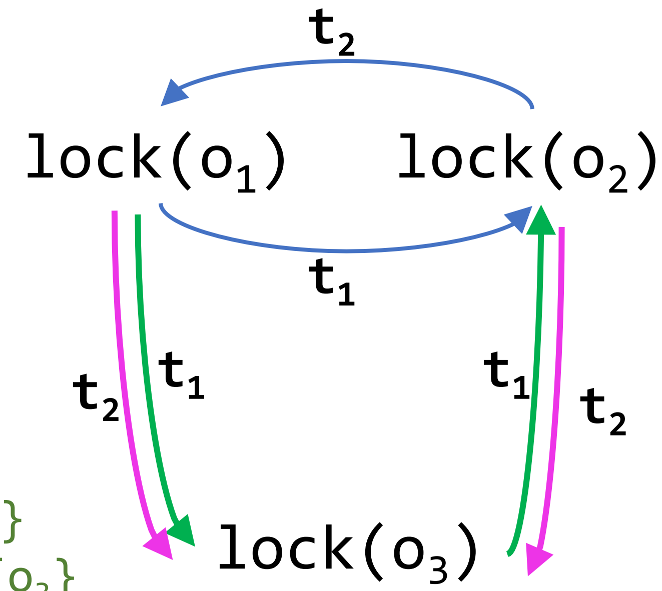
Deadlocks : Lock-dependency Graph

t_1:

```
synchronized(m){//pts(m)={o1}
 synchronized(x){//pts(x)={o3}
  synchronized(n){//pts(n)={o2}
   ...
  }
 }
}
```

t_2:

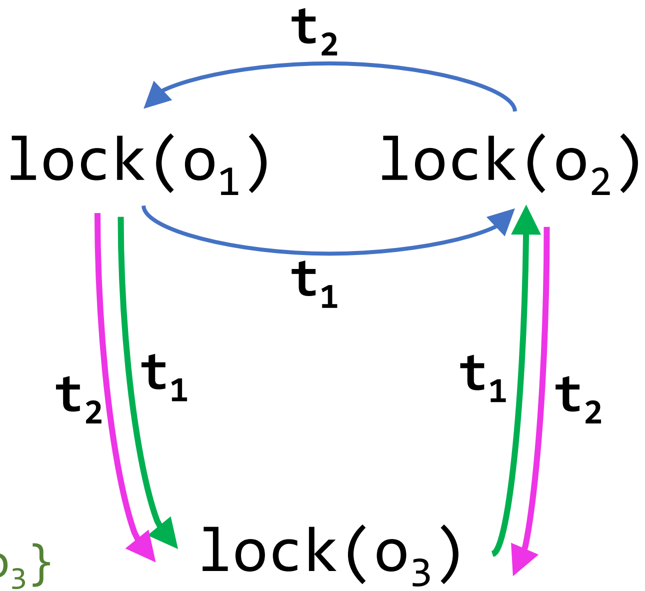
```
synchronized(p){//pts(p)={o2}
   synchronized(q){//pts(q)={o1}
     synchronized(y){//pts(q)={o3}
      ...
     }
   }
}
```



Deadlocks : Lock-dependency Graph

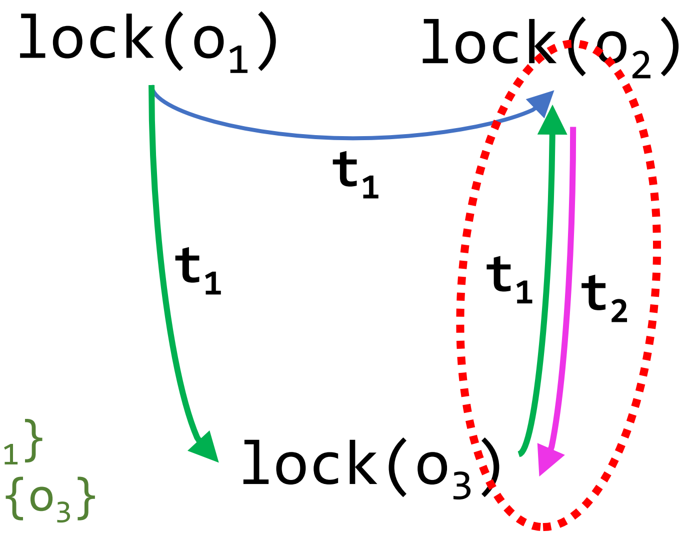
```
t1:
synchronized(m){//pts(m)={o1}
 synchronized(x){//pts(x)={o3}
  synchronized(n){//pts(n)={o2}
   ...
  }
 }
}
```

```
t2:
synchronized(p){//pts(p)={o2}
  synchronized(q){//pts(q)={o1}
    synchronized(y){//pts(q)={o3}
     ...
    }
  }
}
```



Deadlocks : Lock-dependency Graph

```
t1:
synchronized(m){//pts(m)={o1}
 synchronized(x){//pts(x)={o3}
  synchronized(n){//pts(n)={o2}
    ...
  }
 }
}
     t2:
     synchronized(p){//pts(p)={o2}
       synchronized(q){//pts(q)={o1}
         synchronized(y){//pts(q)={o3}
           ...
         }
       }
     }
```



Implementation

D4 is on top of WALA and AKKA for Java programs:

- Client-server
- Eclipse plugin

https://github.com/parasol-aser/D4

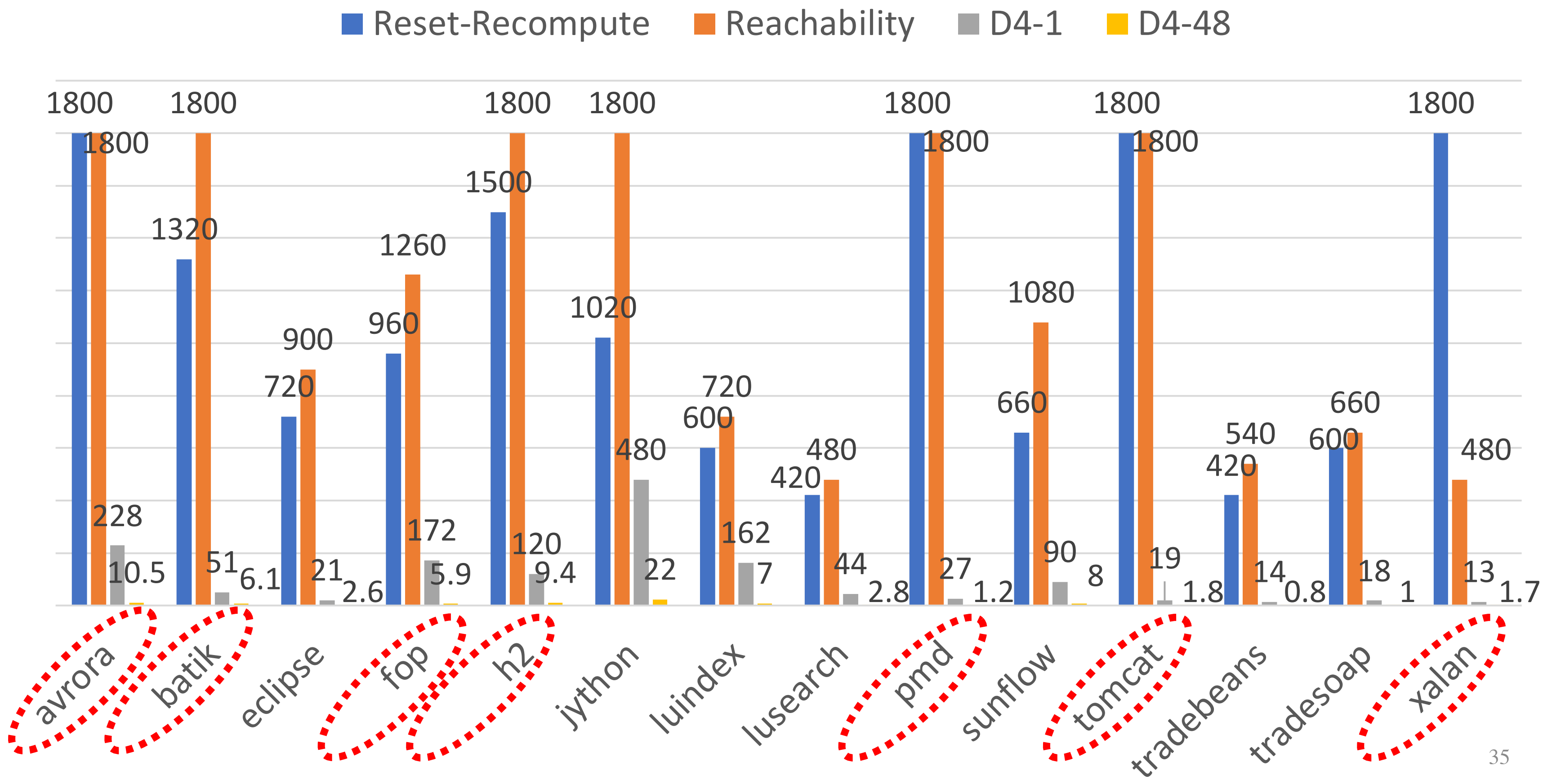
Our pointer analysis has been integrated into WALA:

https://github.com/april1989/Incremental_Points_to_Analysis

Evaluation

- 14 benchmarks in Dacapo-9.12
- Steps:
  1. The whole program detection
  2. Delete Statement
  3. Add Statement
- Performance
  - D4-1 and D4-48
- Precision

Statement Deletion (Worst Cases in Seconds)



Performance: Incremental Pointer Analysis

D4-1 achieves (Incremental):

- 300X speedup on average
- 20X speedup for the worst cases

D4-48 achieves (Parallel & Incremental):

- 3000X speedup on average
- 200X speedup on the worst cases

Performance: Concurrency Bug Detection

D4-48 requires

- 0.12s on average per change
- 20s for the worst cases
- 10X-2000X faster than the exhaustive detection on average
- 5X-50X for the worst cases

Conclusion

D4: a fast framework for concurrency debugging

✓ Two fundamental static analyses:

- Parallel Incremental Pointer Analysis
- Parallel Incremental Static Happens-before Analysis

✓ Pinpoint bugs within 100ms on average after a code change

✓ 20X-1000X faster than the exhaustive and incremental analyses

D4:
Fast Concurrency Debugging with Parallel Differential Analysis

https://github.com/parasol-aser/D4

Thank you!

Bozhen Liu **Jeff Huang**

Texas A&M University