OoOJava: An Out-of-Order Approach to Parallel Programming

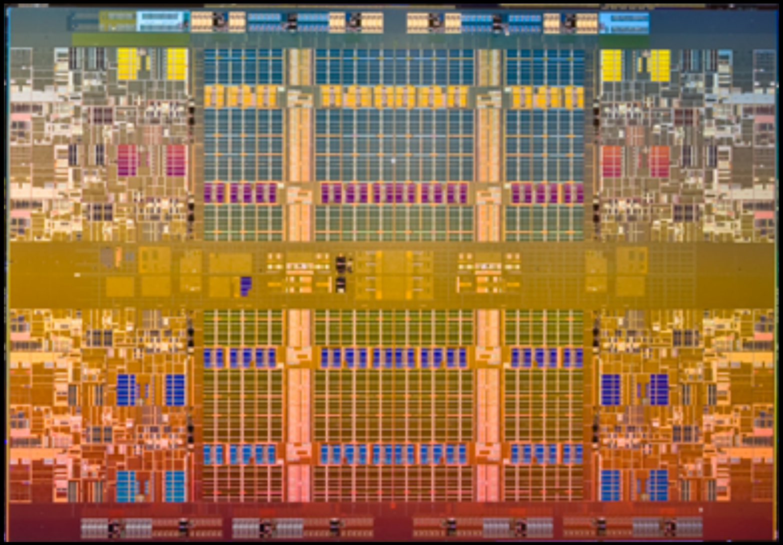
Jim Jenista

Yong hun Eom

Brian Demsky

University of California, Irvine

Motivation



- Parallel software development is difficult
- Locks are prone to races and deadlocks
- Concurrency bugs are hard to find and fix

➡ **Need easier model**

Out-of-Order Java (OoOJava)

- OoOJava inspired by superscalar processors
- Extends Java with re-orderable block (rblock)
- Annotation decouples block from main thread
- Preserves sequential semantics

➡ **Annotation errors do not affect correctness only performance**

Code Example

```
while( methodsItr.hasNext() ) {
    m = methodsItr.next();
    ast = d2ast.get( m );

    ast.typeCheck();
    cfg = ast.flatten();


    d2cfg.put( m, cfg );

}
d2cfg.serializeToDisk();
```

Code Example

```
while( methodsItr.hasNext() ) {
    m = methodsItr.next();
    ast = d2ast.get( m );
    rblock p {
        ast.typeCheck();
        cfg = ast.flatten();
    }

    d2cfg.put( m, cfg );

}
d2cfg.serializeToDisk();
```

parent rblock

p

Code Example

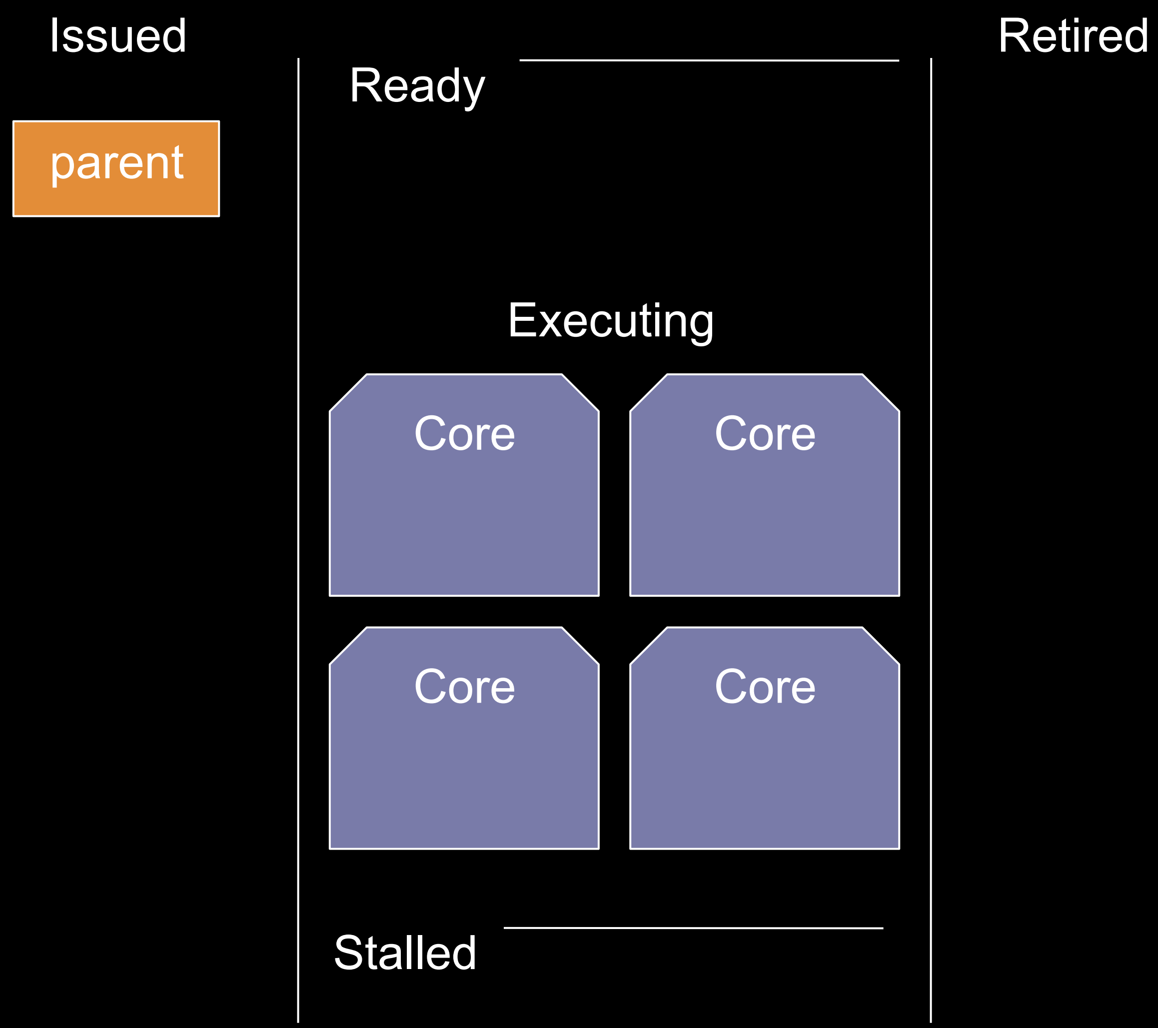
```
while( methodsItr.hasNext() ) {
      m = methodsItr.next();
      ast = d2ast.get( m );
    rblock p {
          ast.typeCheck();
          cfg = ast.flatten();
    }
    rblock s {
          d2cfg.put( m, cfg );
    }
}
d2cfg.serializeToDisk();
```

parent rblock

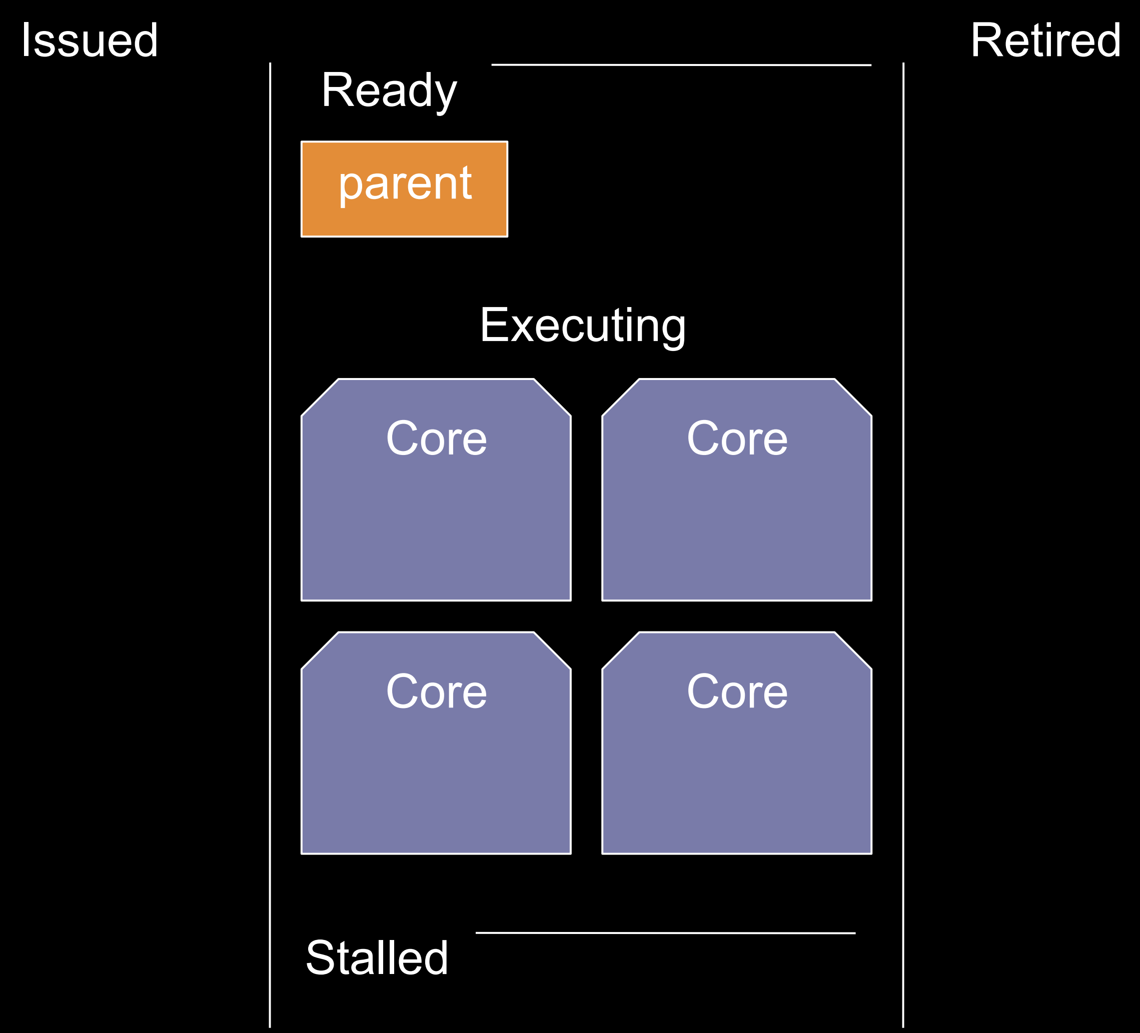
p

s

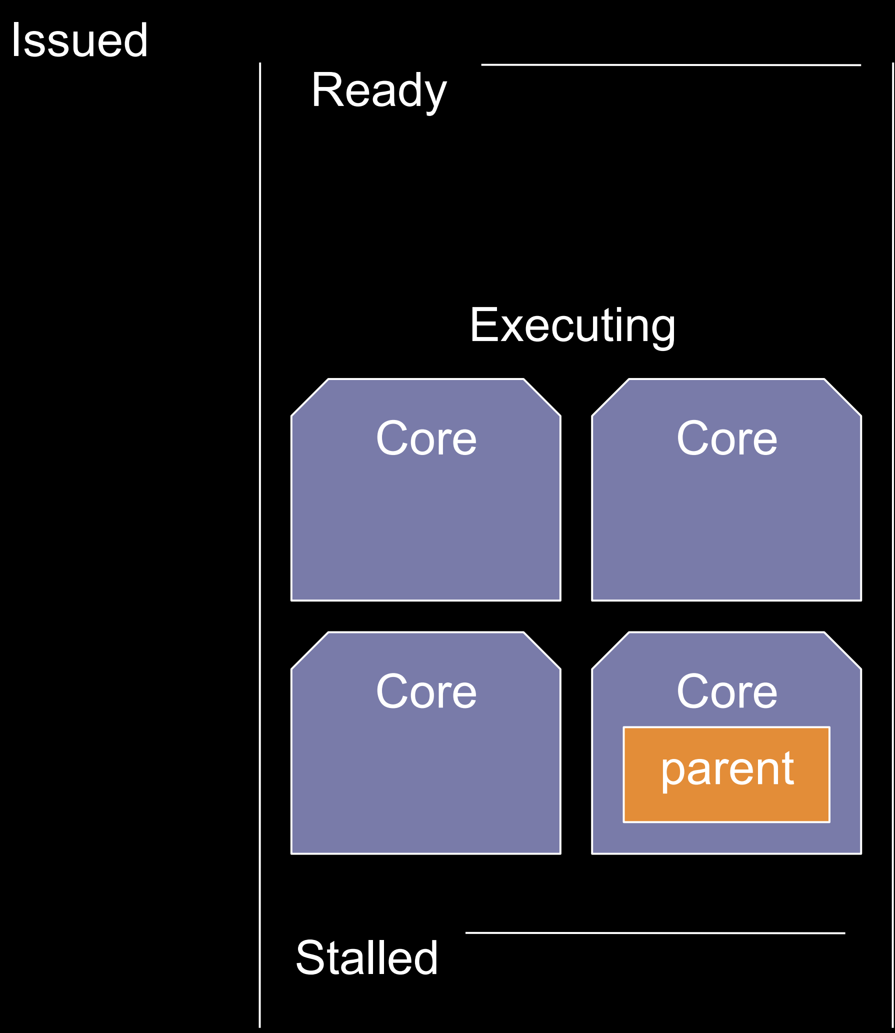
Out-of-Order Java Execution



Out-of-Order Java Execution



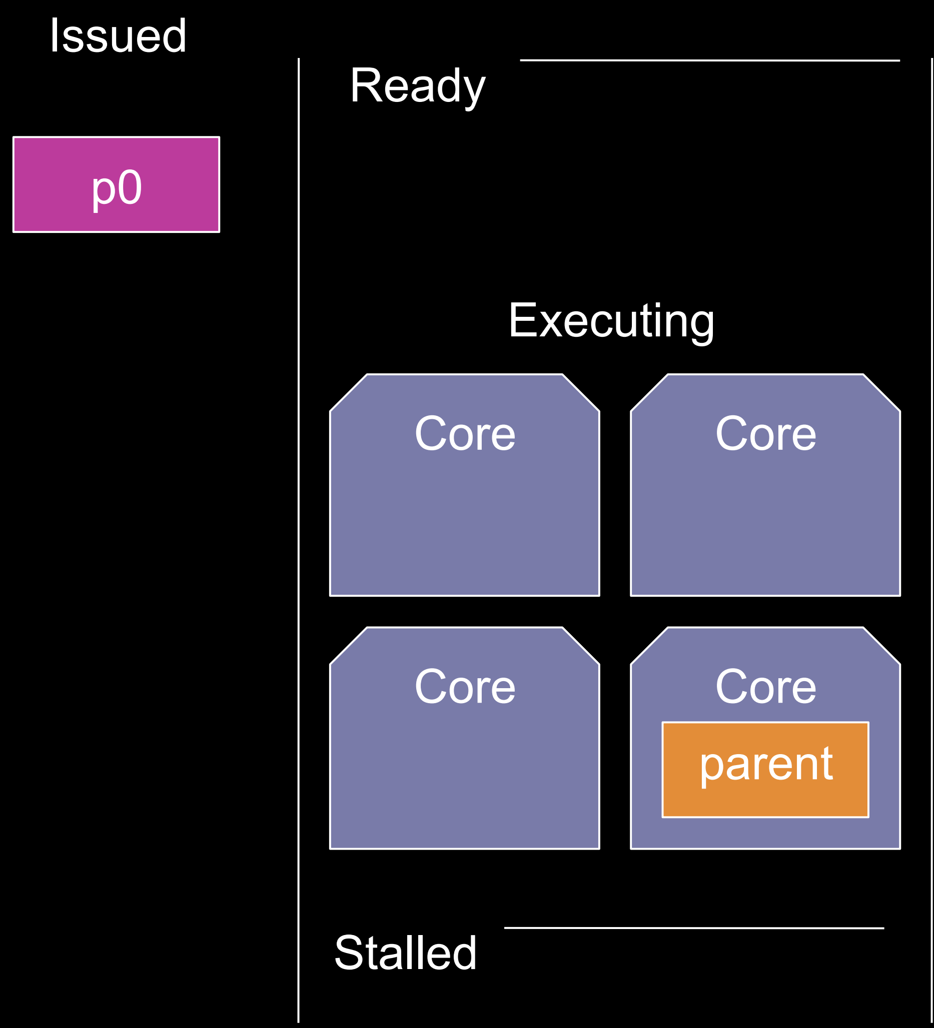
Out-of-Order Java Execution



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution

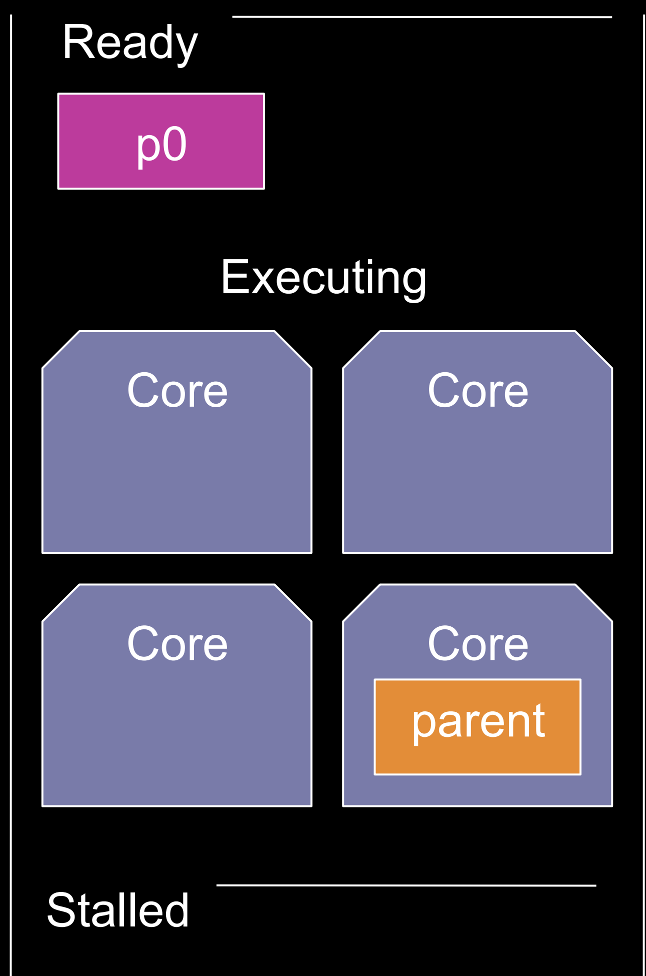


Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution

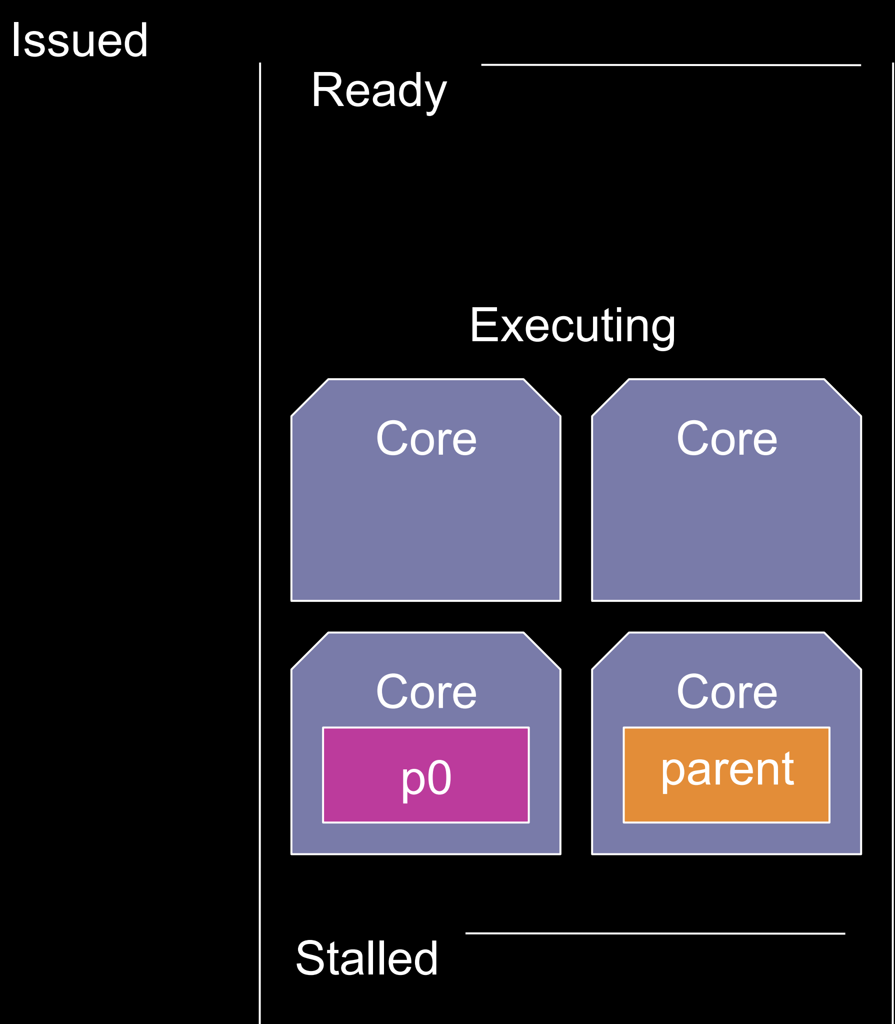
Issued



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

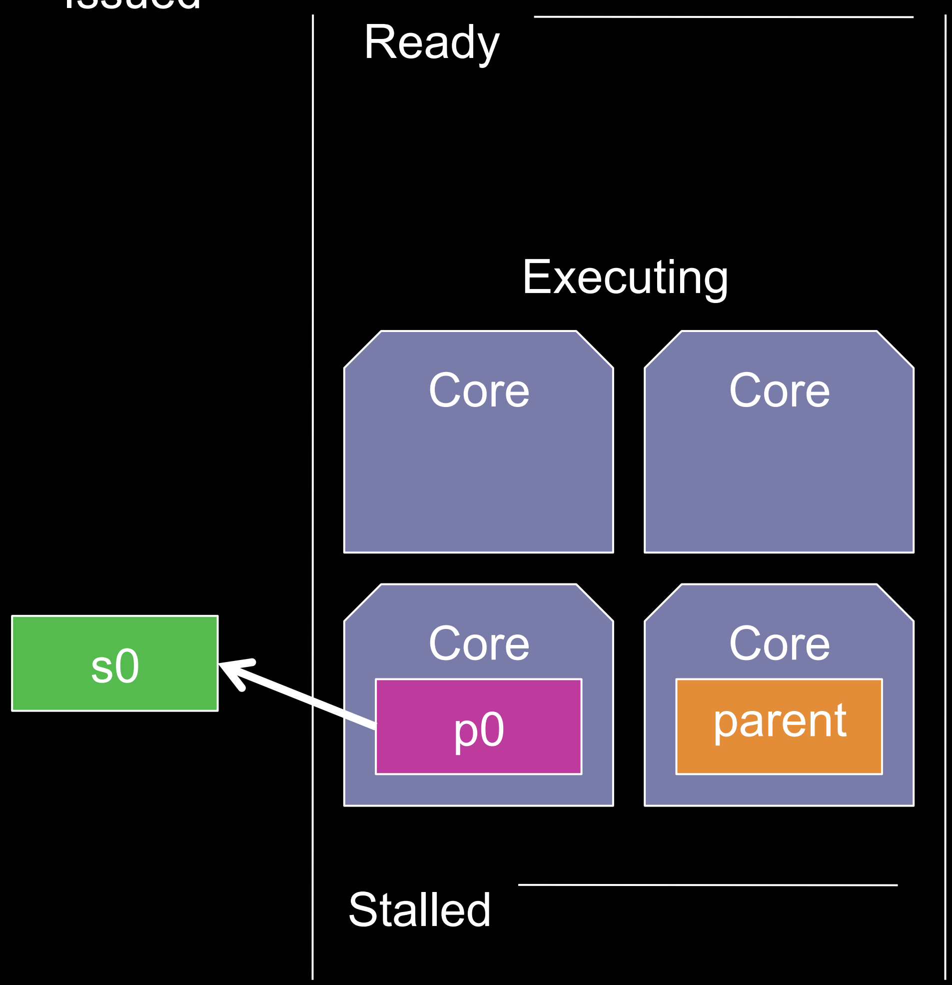
Out-of-Order Java Execution



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

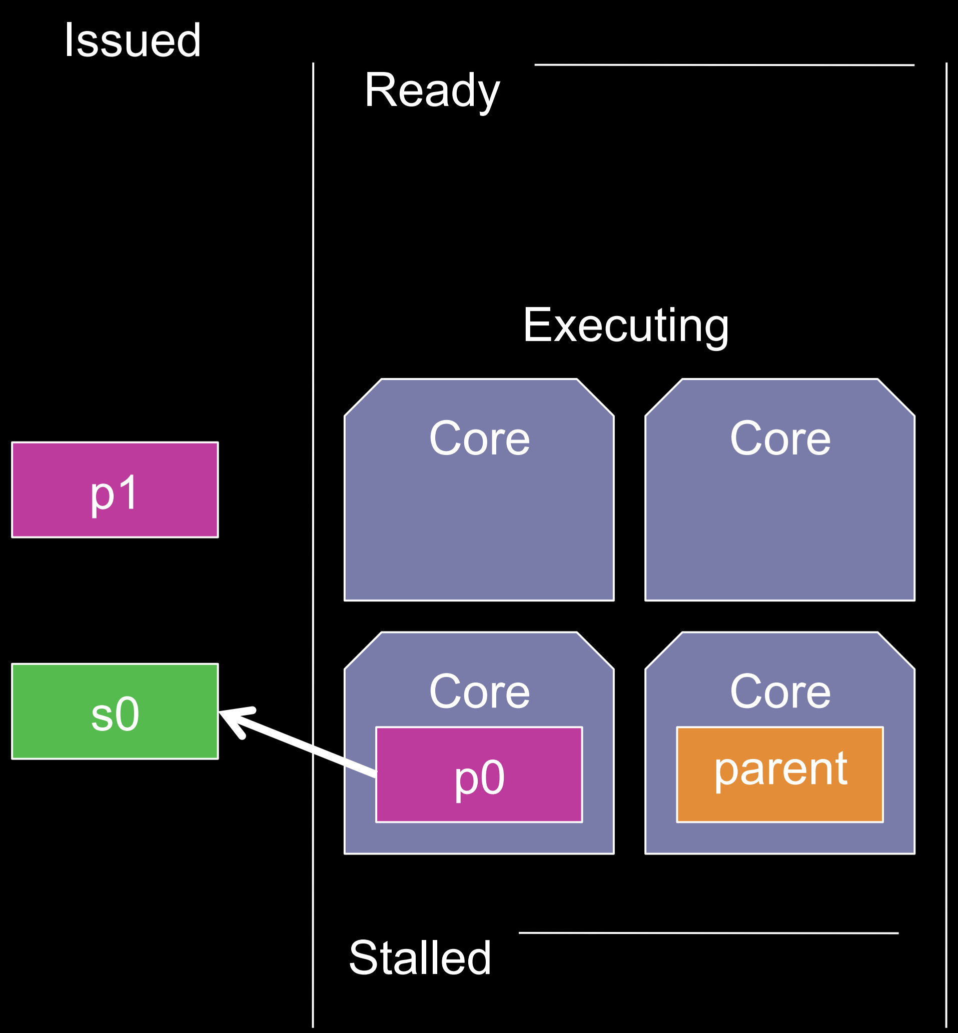
Out-of-Order Java Execution



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

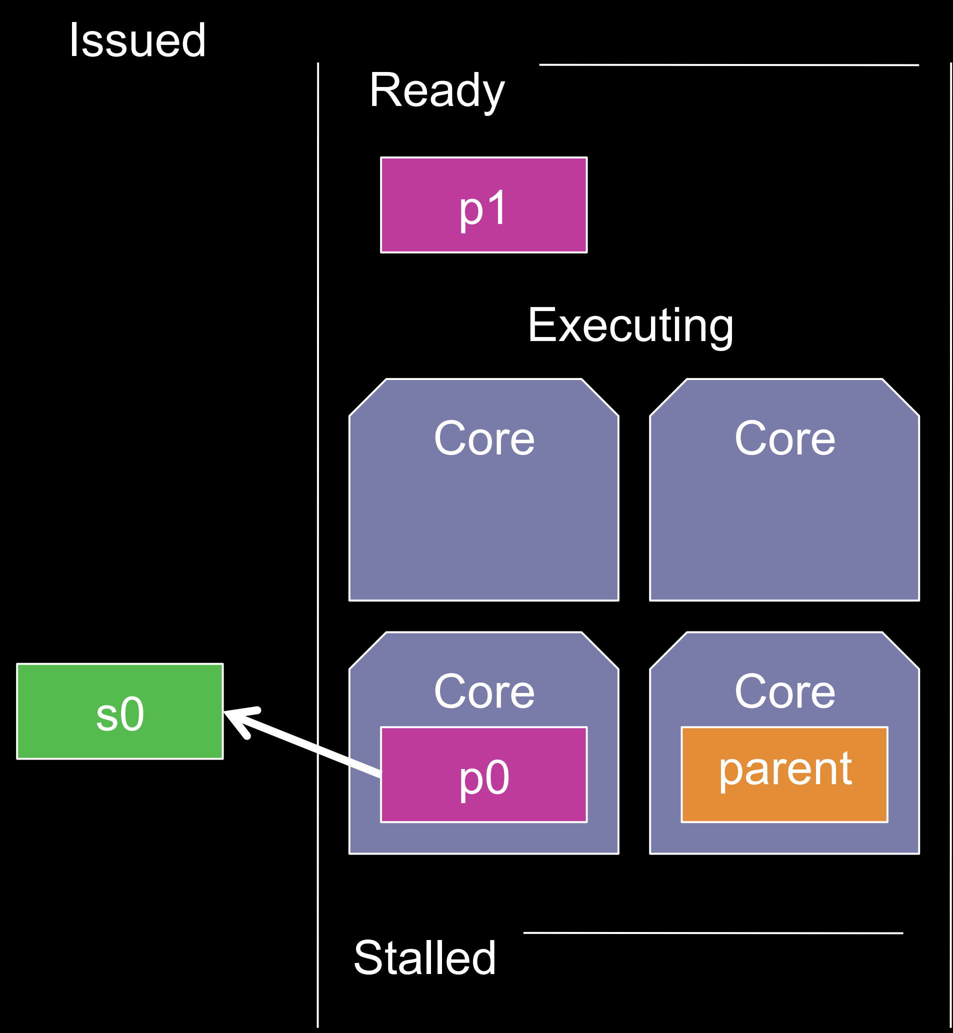
Out-of-Order Java Execution



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

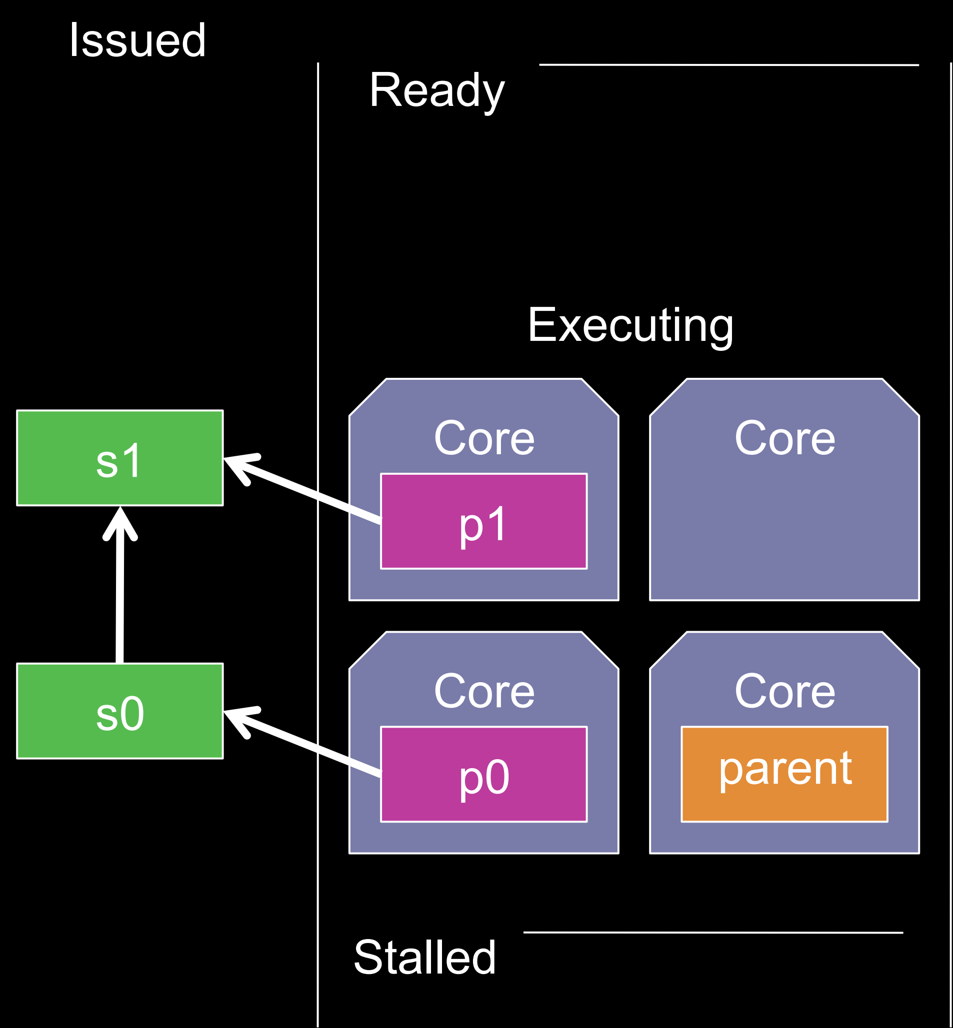
Out-of-Order Java Execution



Retired

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

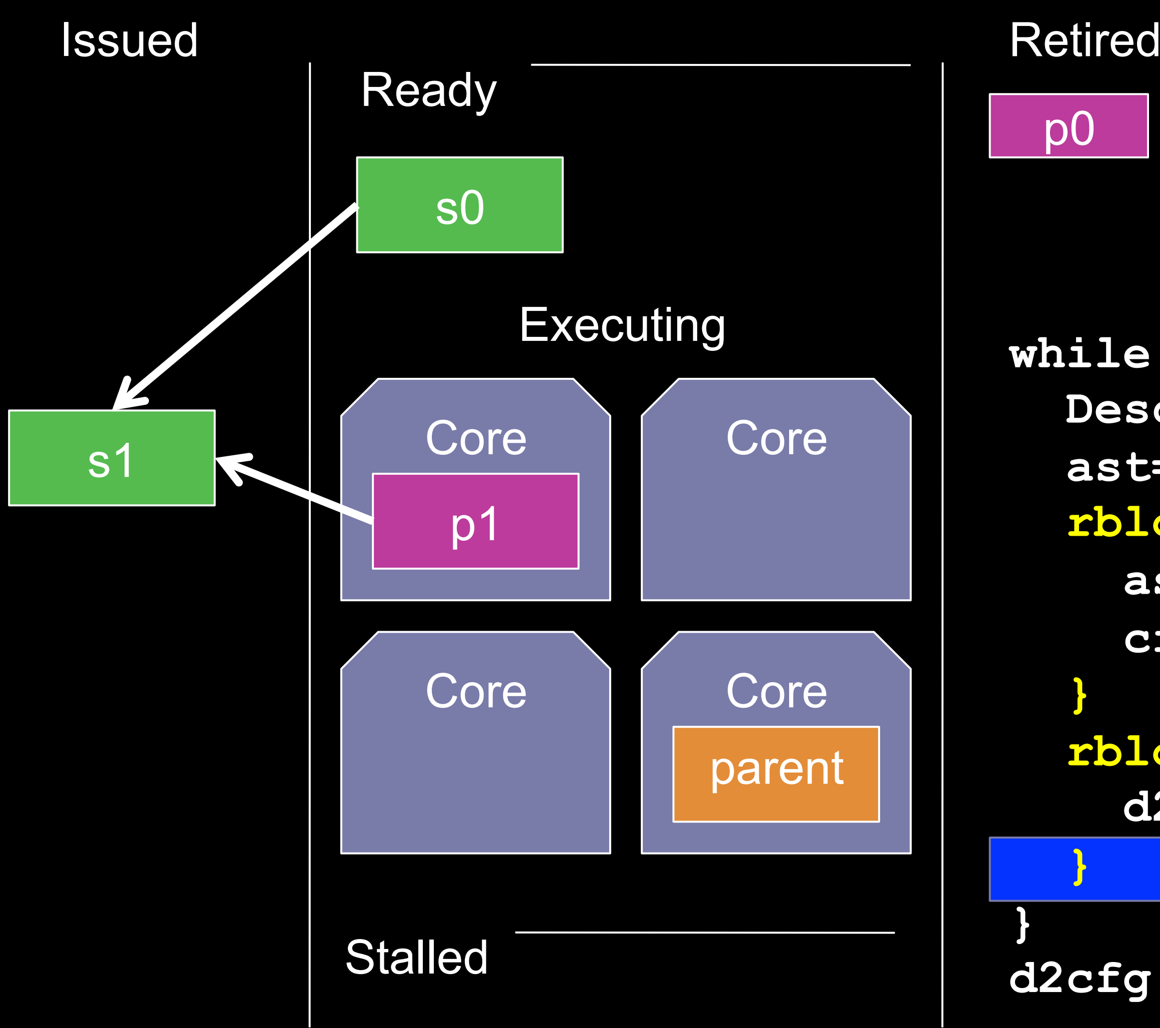
Out-of-Order Java Execution



Retired

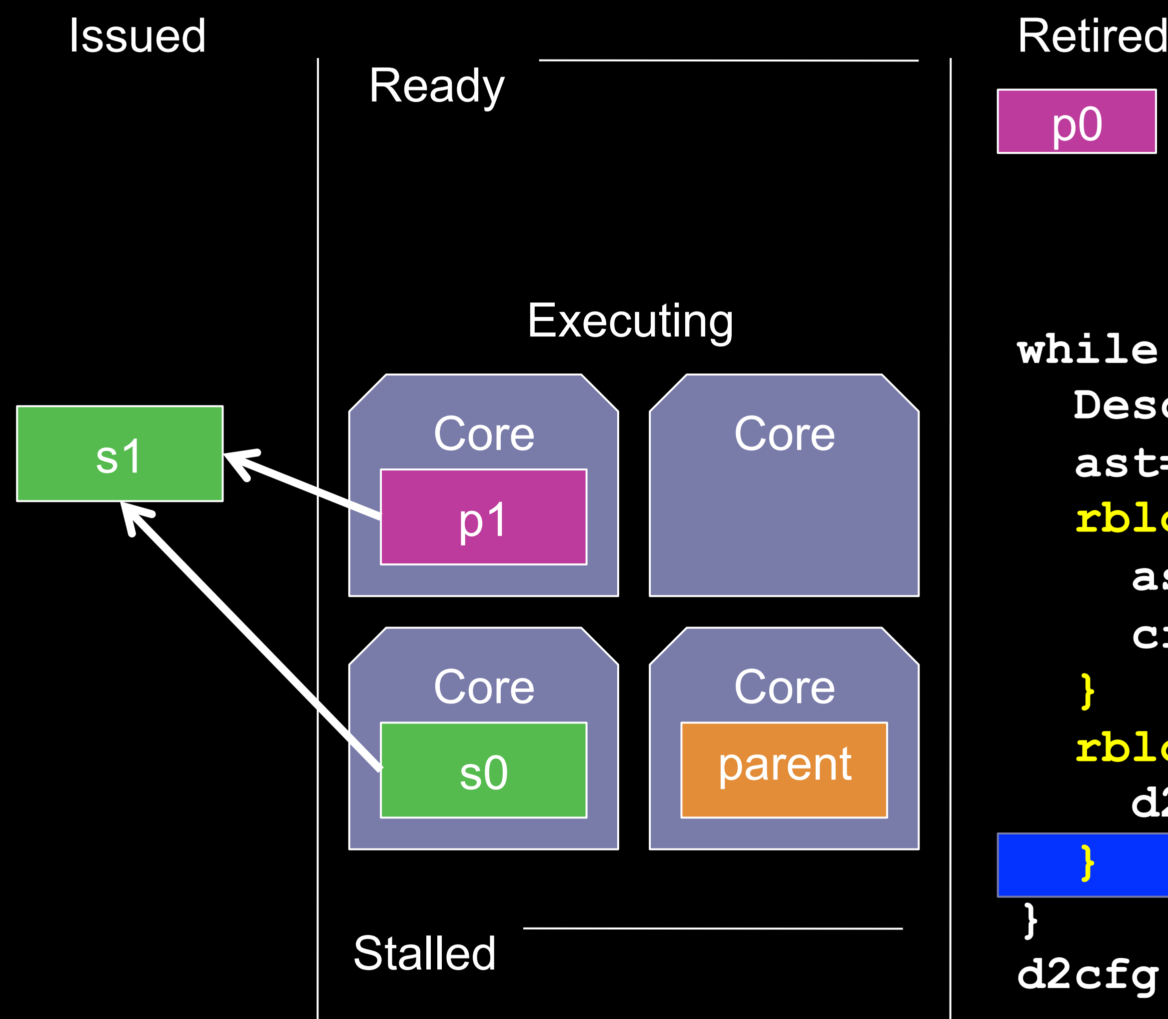
```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution



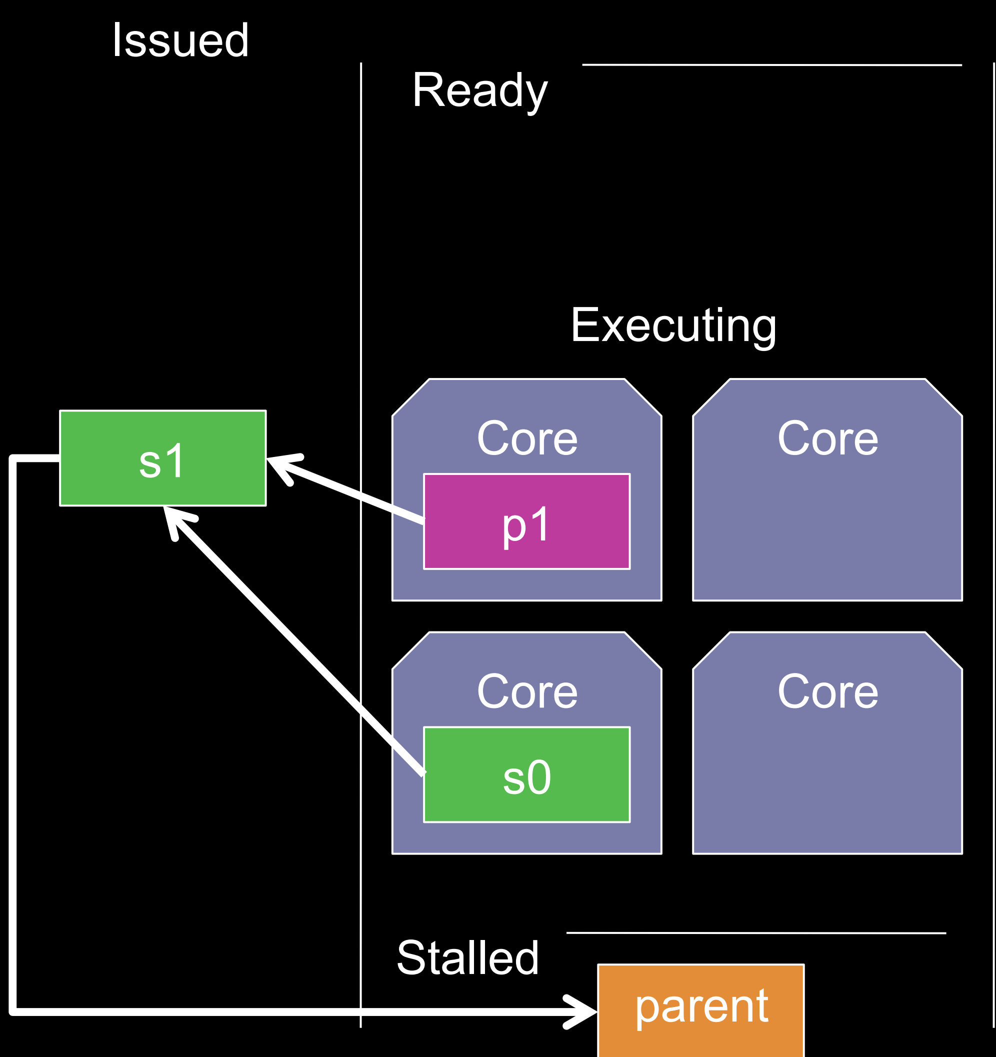

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution



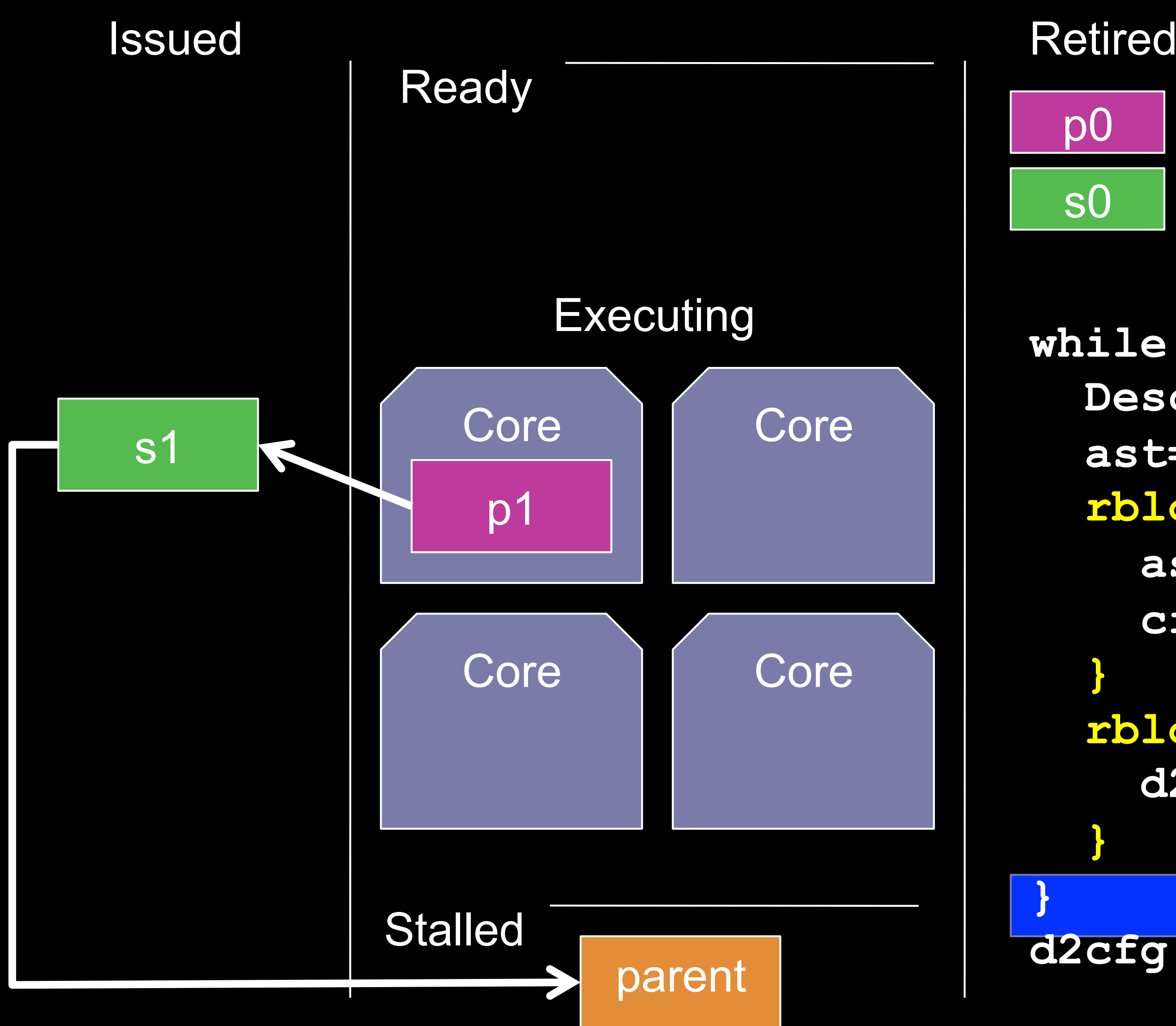

```
while( ... ) {
   Descriptor m= ...;
   ast=d2ast.get( m );
   rblock p {
      ast.typeCheck();
      cfg=ast.flatten();
   }
   rblock s {
      d2cfg.put( m, cfg );
   }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution



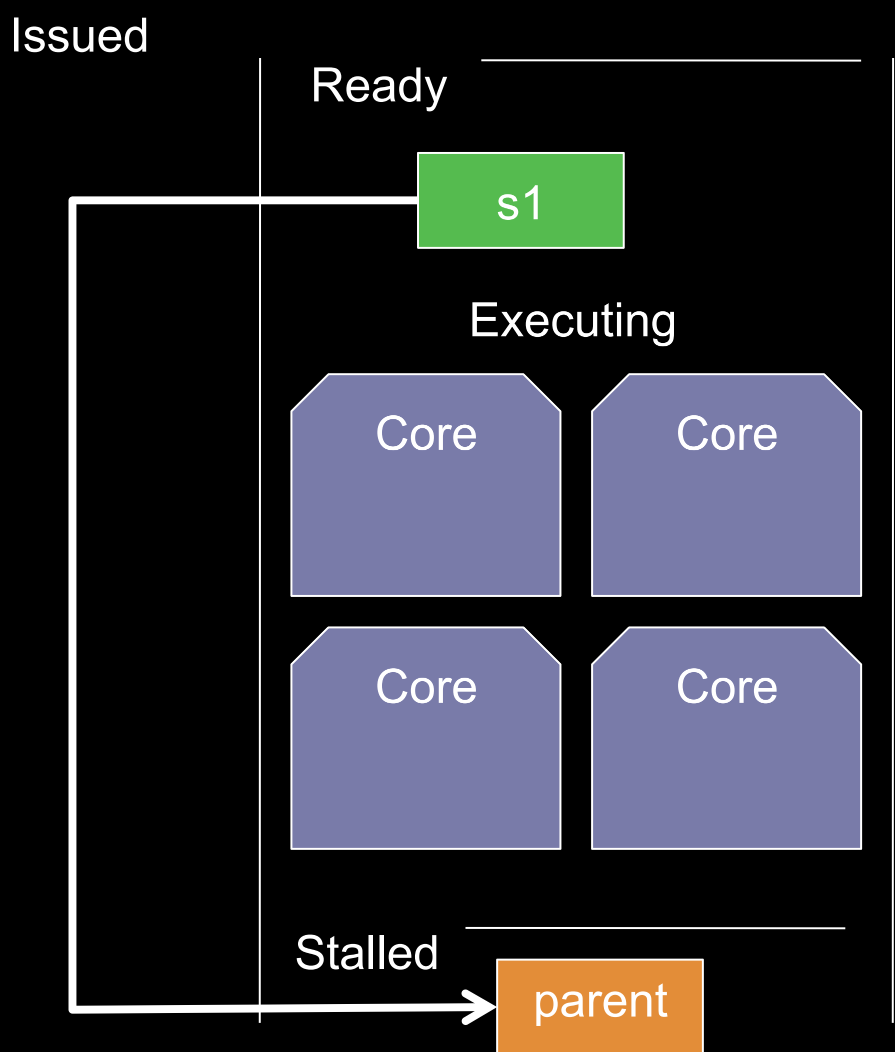

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution



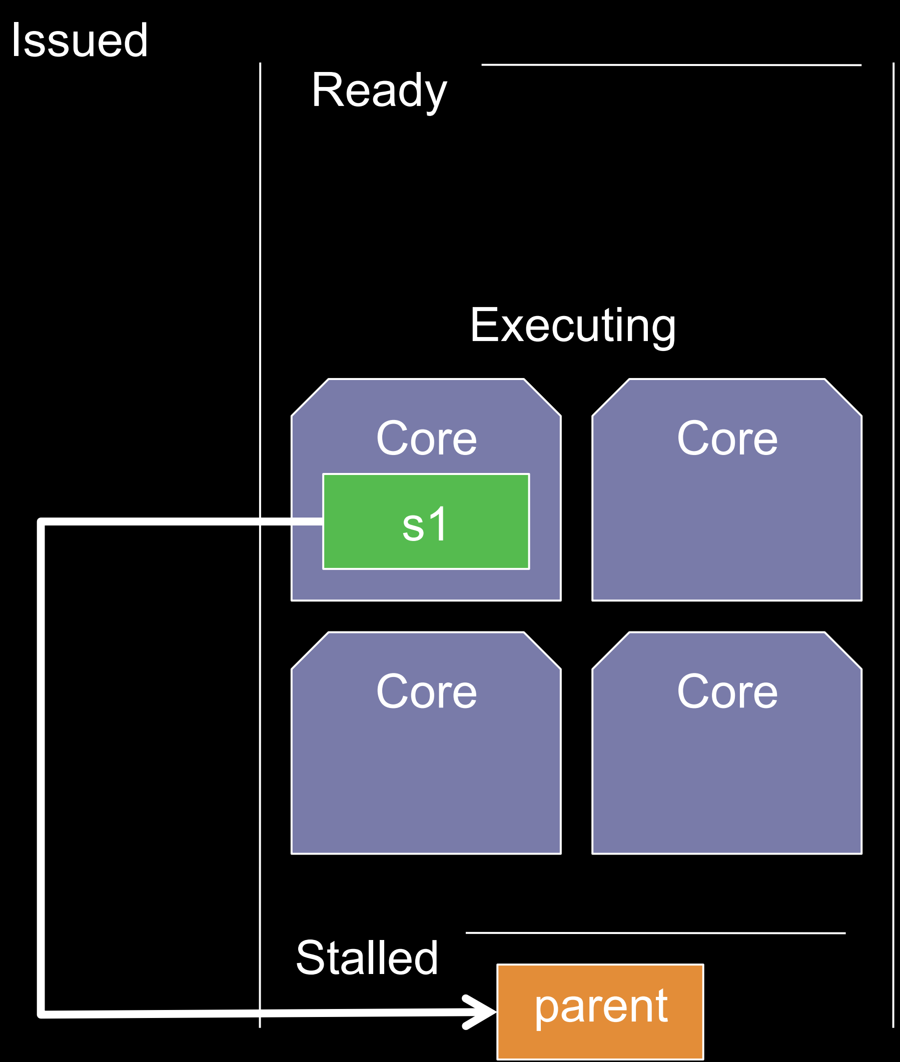

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

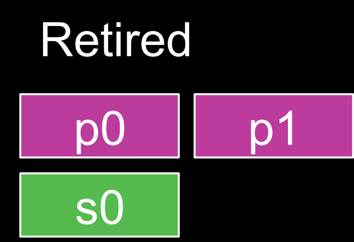
Out-of-Order Java Execution




```
while( ... ) {
   Descriptor m= ...;
   ast=d2ast.get( m );
   rblock p {
      ast.typeCheck();
      cfg=ast.flatten();
   }
   rblock s {
      d2cfg.put( m, cfg );
   }
}
d2cfg.serializeToDisk();
```

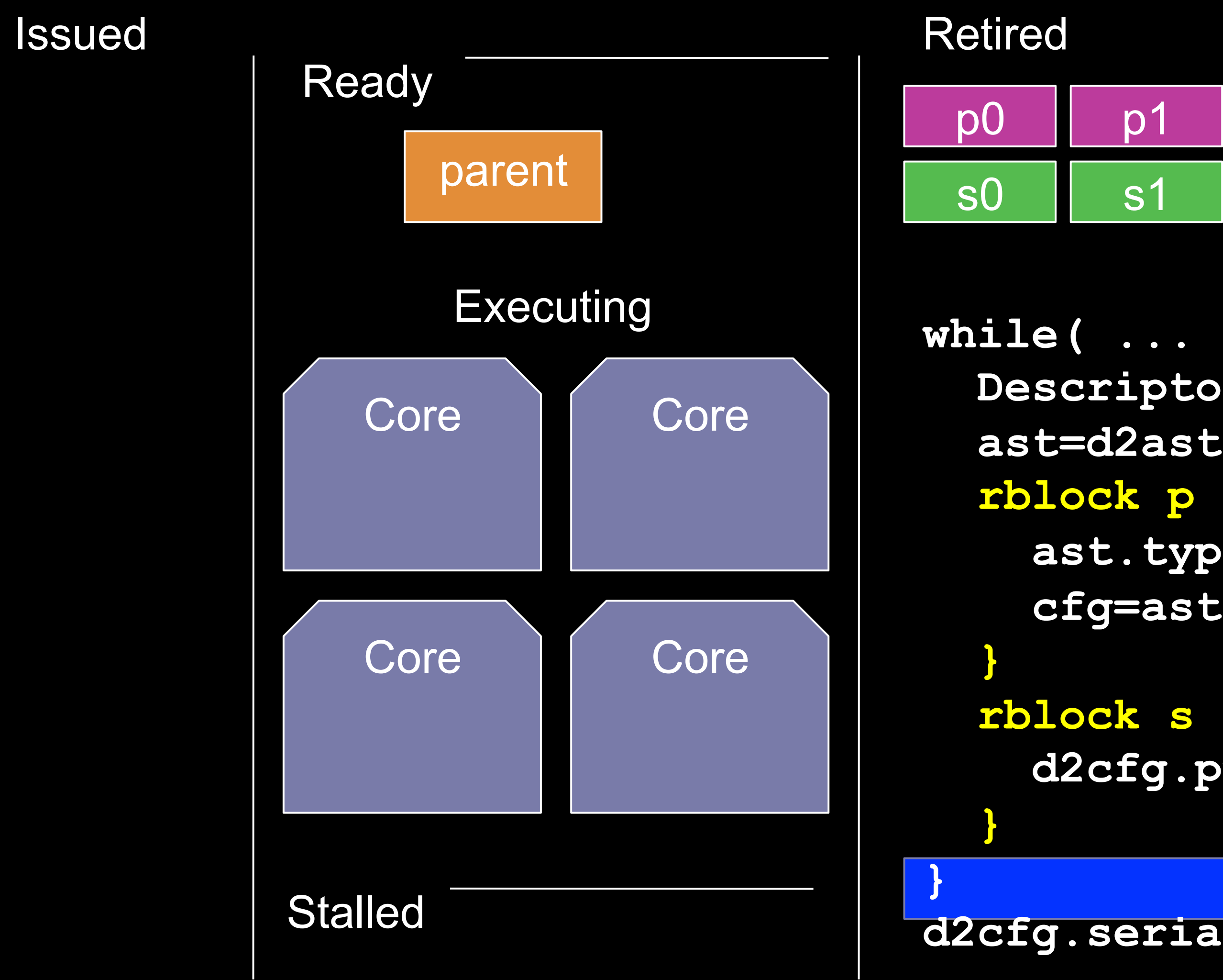
Out-of-Order Java Execution





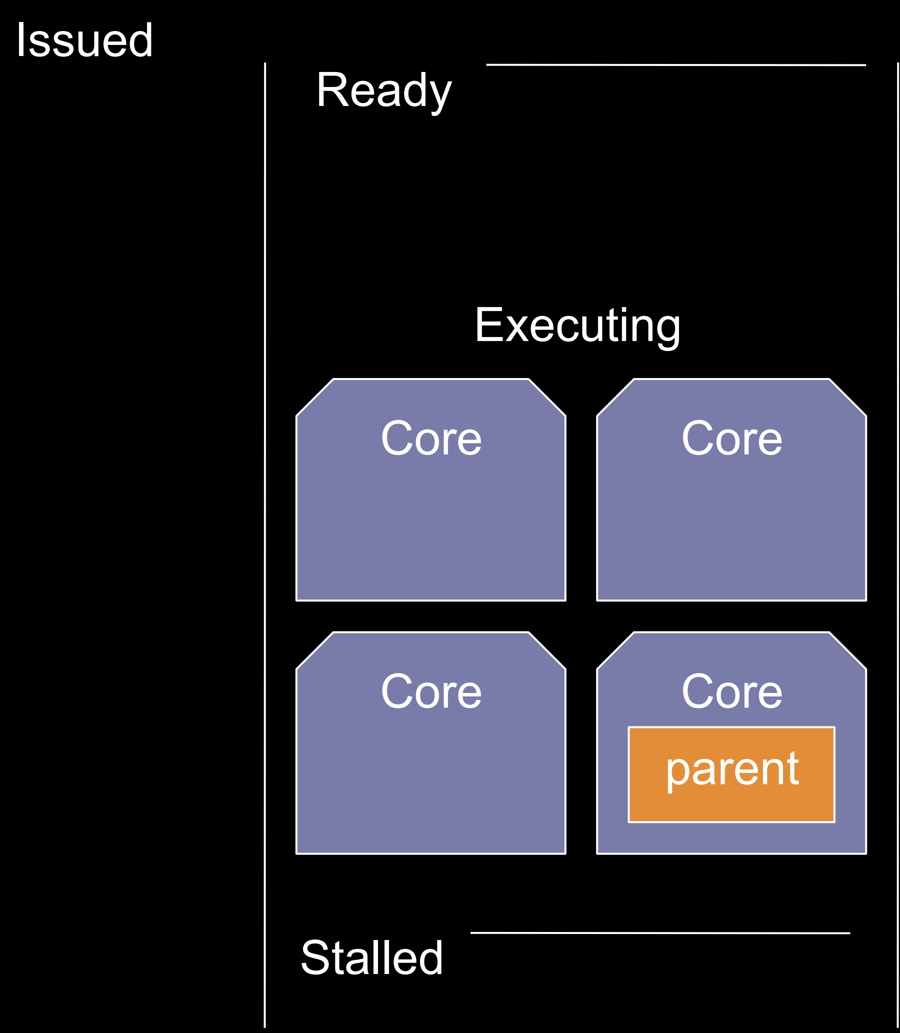

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution



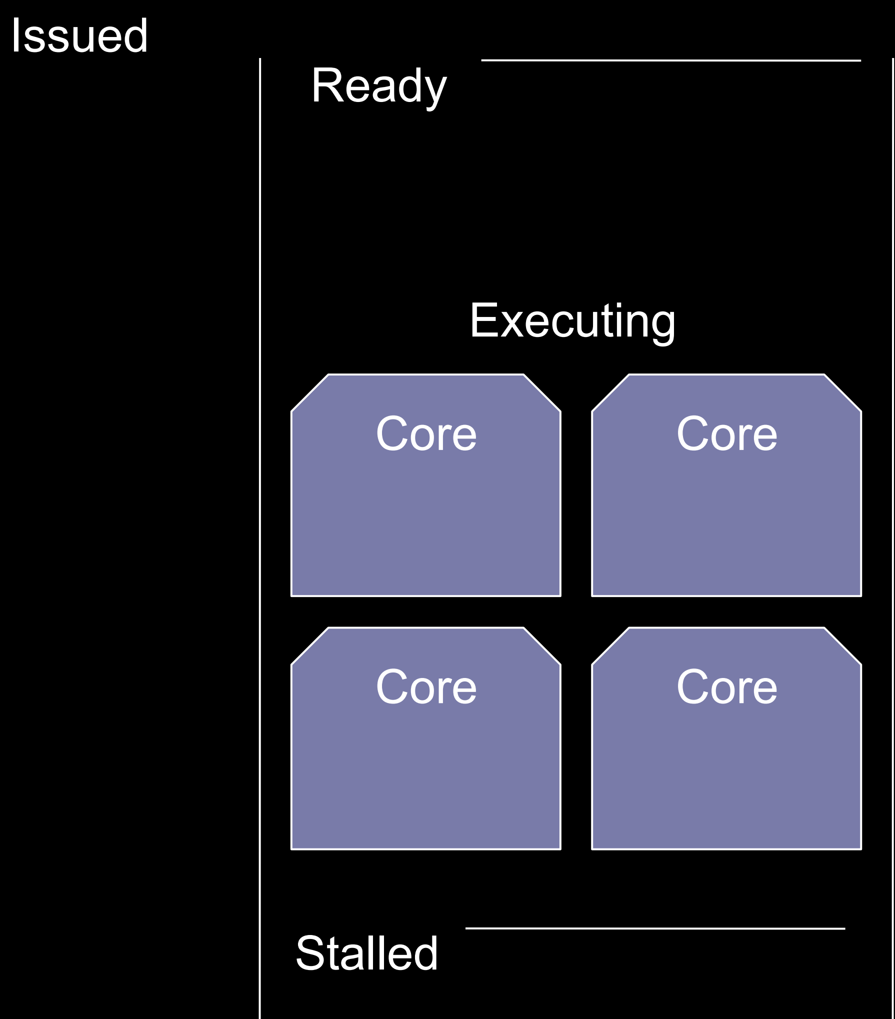

```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

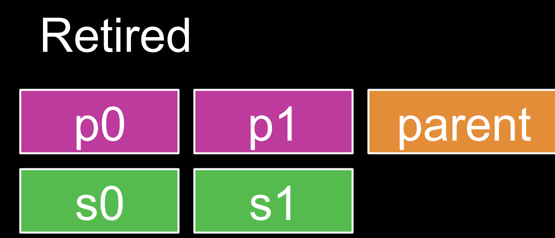
Out-of-Order Java Execution




```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Out-of-Order Java Execution






```
while( ... ) {
  Descriptor m= ...;
  ast=d2ast.get( m );
  rblock p {
    ast.typeCheck();
    cfg=ast.flatten();
  }
  rblock s {
    d2cfg.put( m, cfg );
  }
}
d2cfg.serializeToDisk();
```

Reorderable Block Hierarchy

- Reorderable blocks support arbitrary composition including nesting

```
rblock a {
   …
   rblock b {
   }
}
```

- rblock instances form a hierarchy at runtime

Execution Semantics

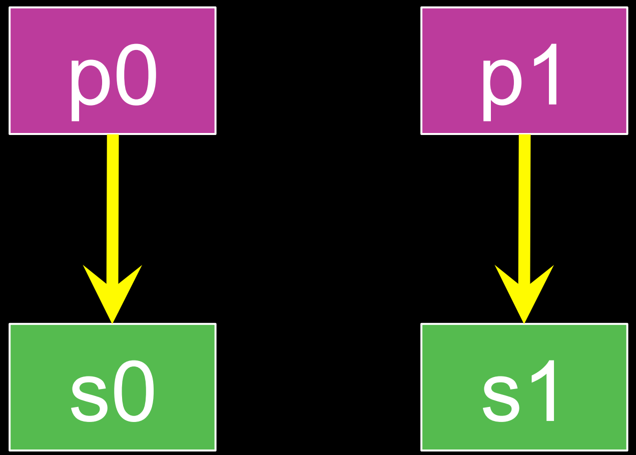
- Respects dependences between rblocks
- Control dependences handled implicitly
- Two types of data dependences:
  - Variable dependences
  - Heap dependences

OoOJava safely approximates all data dependences automatically

Variable Dependence Analysis

WAW, WAR Hazards

```
rblock p {
  ast.typeCheck();
  cfg = ast.flatten();
}
rblock s {
  d2cfg.put( m, cfg );
}
```



- Many instances of an rblock may be in-flight
- Forwarded values eliminate Write-after-Write and Write-after-Read hazards

➡ Track dependences and forward values

In-set, Out-set Variables

```
rblock p {
   ast.typeCheck();
   cfg = ast.flatten();
}
rblock s {
   d2cfg.put( m, cfg );
}
```

rblock p:
in-set = {ast}
out-set = {cfg}

rblock s:
in-set = {d2cfg, m , cfg}
out-set = { }

Variable Source Analysis

```
rblock p {
  ast.typeCheck();
  cfg = ast.flatten();
}
rblock s {
  d2cfg.put( m, cfg );
}
```

$\text{cfg} \rightarrow \{ \langle p, 0, \text{cfg} \rangle \}$

Writing Variables

```
rblock r {
    rblock c { x = v.f; }
```

$x \to \{ \langle c, 0, x \rangle \}$

Kill facts for x

```
    x = w.g;
```

$x \to \{ \langle r, 0, x \rangle \}$

```
}
```

Reading Variables

```
rblock r {
    rblock c { x = v.f; }
```

$x \rightarrow \{ \langle c, 0, x \rangle \}$

Kill facts for x

```
    … = x.g;
```

$x \rightarrow \{ \langle r, 0, x \rangle \}$

```
}
```

Avoiding Stalls

```
rblock r {
    rblock c { y = v.f; }
```

$y \rightarrow \{ \langle c, 0, y \rangle \}$

Kill facts for x

```
    x = y;
```

$x \rightarrow \{ \langle c, 0, y \rangle \}, y \rightarrow \{ \langle c, 0, y \rangle \}$

```
}
```

rblock enter

$$x \rightarrow \{ \langle r, 0, x \rangle \}$$

$$y \rightarrow \{ \langle q, 0, y \rangle \}$$

```
rblock r {
```

$$x \rightarrow \{ \langle r, 1, x \rangle \}$$

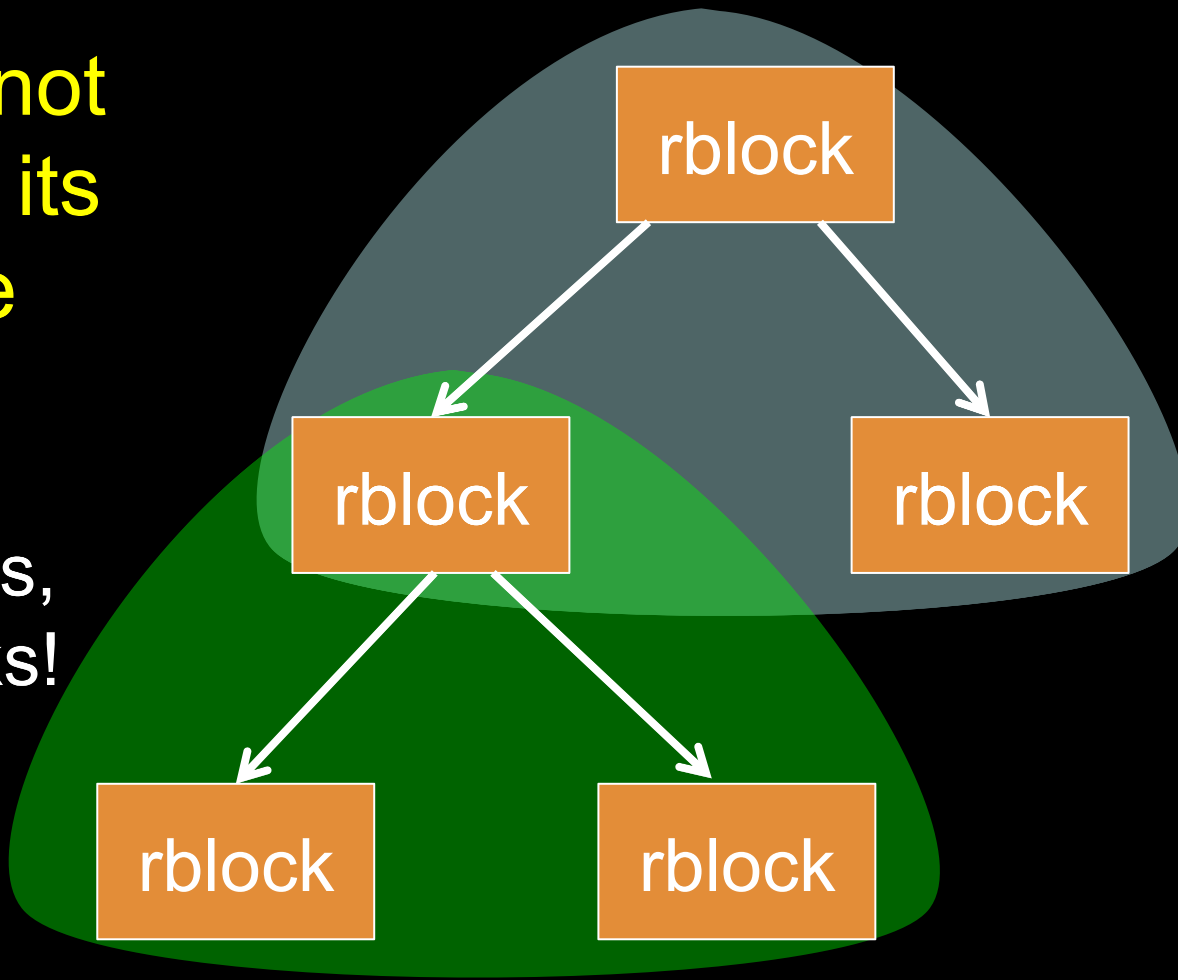
$$y \rightarrow \{ \langle q, 0, y \rangle \}$$

```
}
```

Dependence Structure

A parent cannot retire until all its children have retired

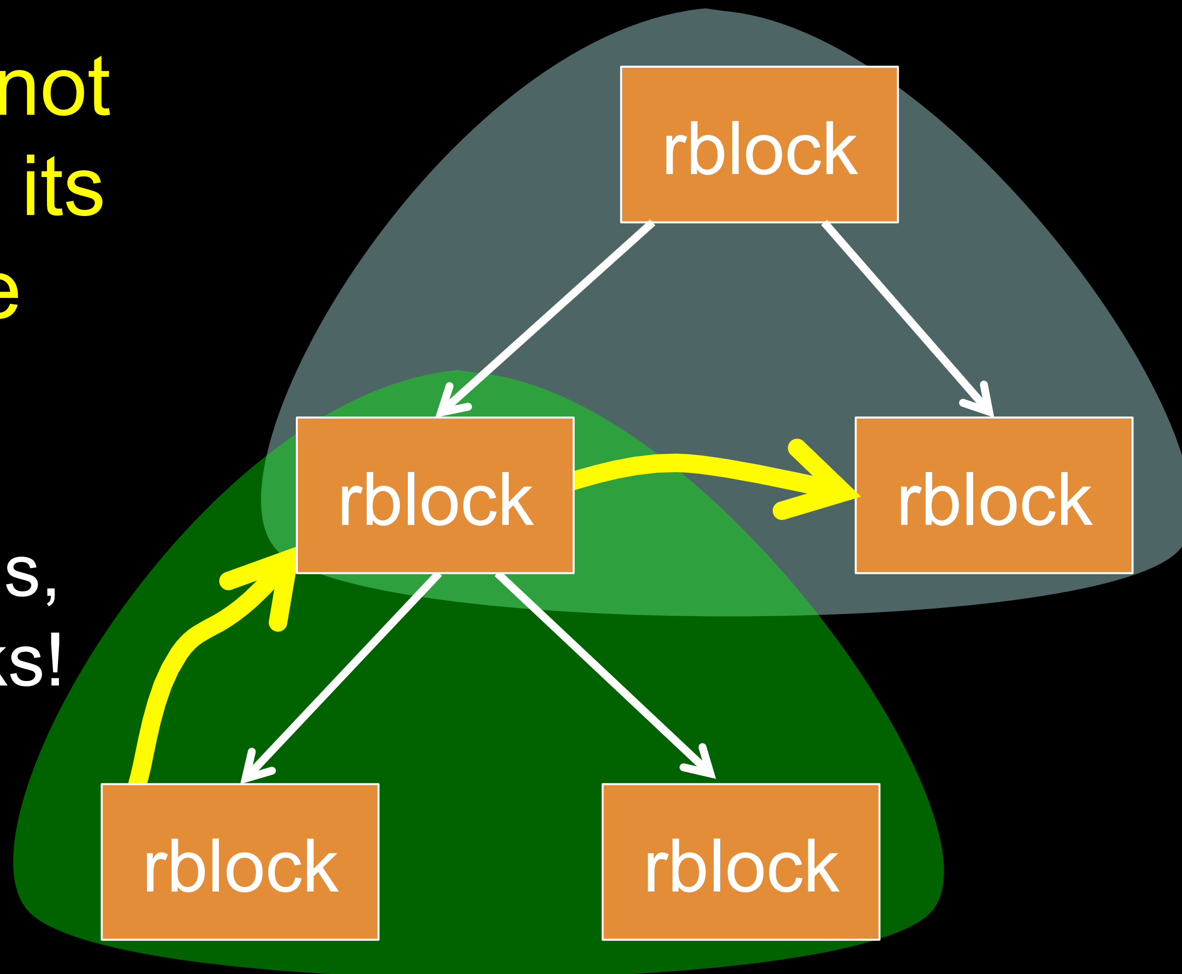
Bad for humans, good for rblocks!



Dependence Structure

A parent cannot retire until all its children have retired

Bad for humans, good for rblocks!



rblock exit

```
rblock r {
   rblock c { x = v.f; }
```

$x \rightarrow \{ \langle c, 0, x \rangle \}$

```
}
```

$x \rightarrow \{ \langle r, 0, x \rangle \}$

Generating Variable Accesses

- Variable's sources are from current rblock, its ancestors, or their siblings

  ⇒ Value is available

- Variable's source is a single tuple from a child c with age $a < k$

  ⇒ Stall for a^{th} oldest instance of c, get value forwarded

Generating Variable Accesses

- Cannot statically resolve variable source

⇨ Generate code to dynamically track source

Outcomes when dynamic variable accessed:

- Variable may reference value

or

- Variable may reference rblock instance, stall for rblock and get value forwarded

Heap Dependence Analysis

Heap Dependences

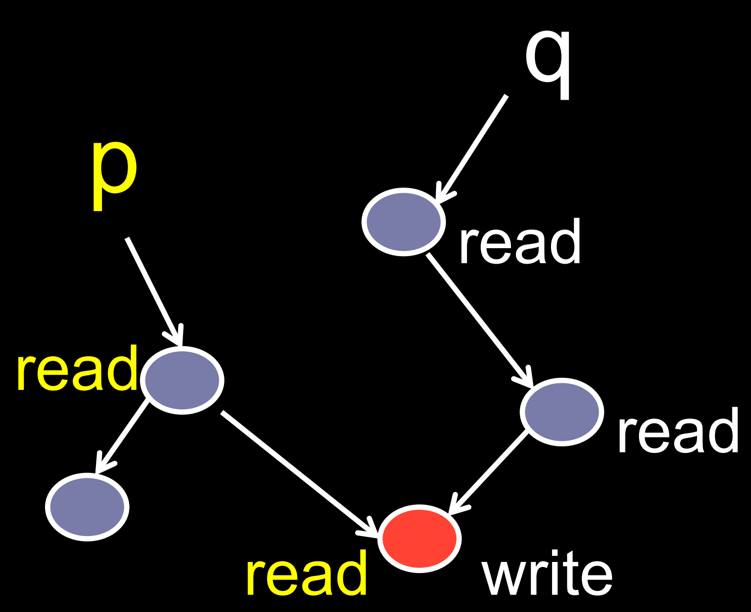
Must ensure that:

(1) all writes to a memory location occur in the same order and

(2) reads from a memory location execute between the same writes as the sequential execution.

A Brute Force Approach

- To issue rblock p, for each previous rblock q that has not retired:



A Brute Force Approach

- To is… ue rblock p… for ea… previous rblock q that has… reti…

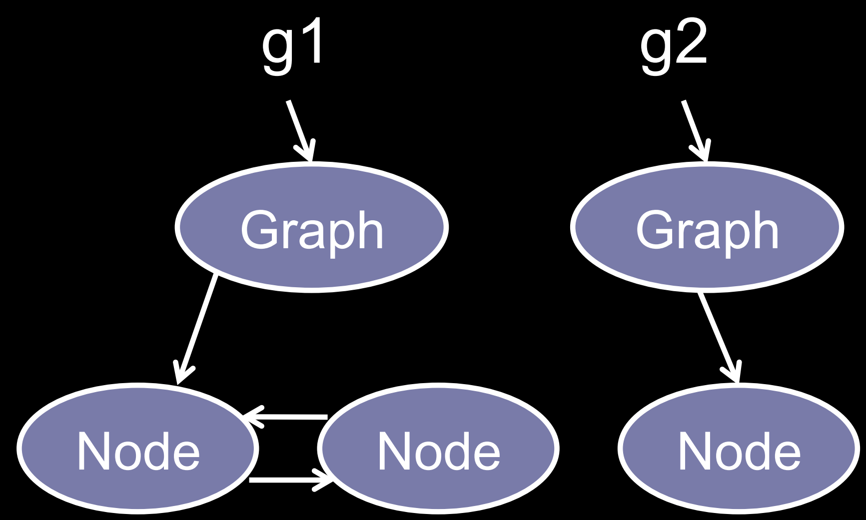
TOO EXPENSIVE!

Solution: Use Reachability

- Every Node in heap is reachable from at most one Graph object

$+$

- p's g1 $\neq$ q's g2

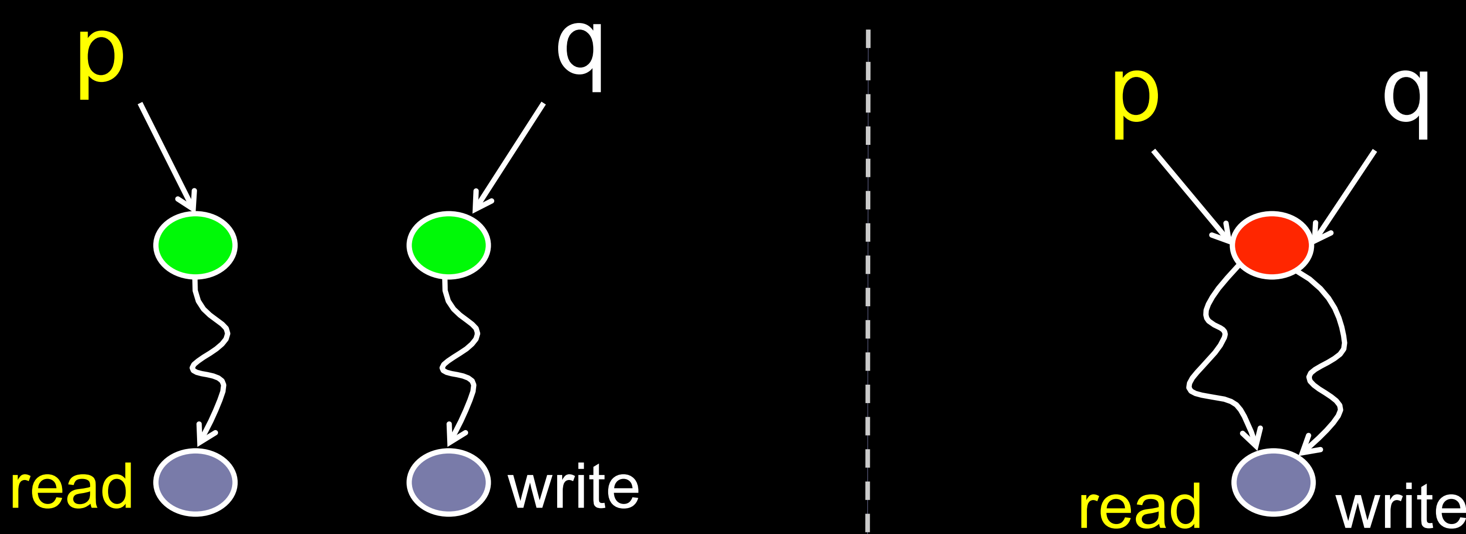


$\Downarrow$

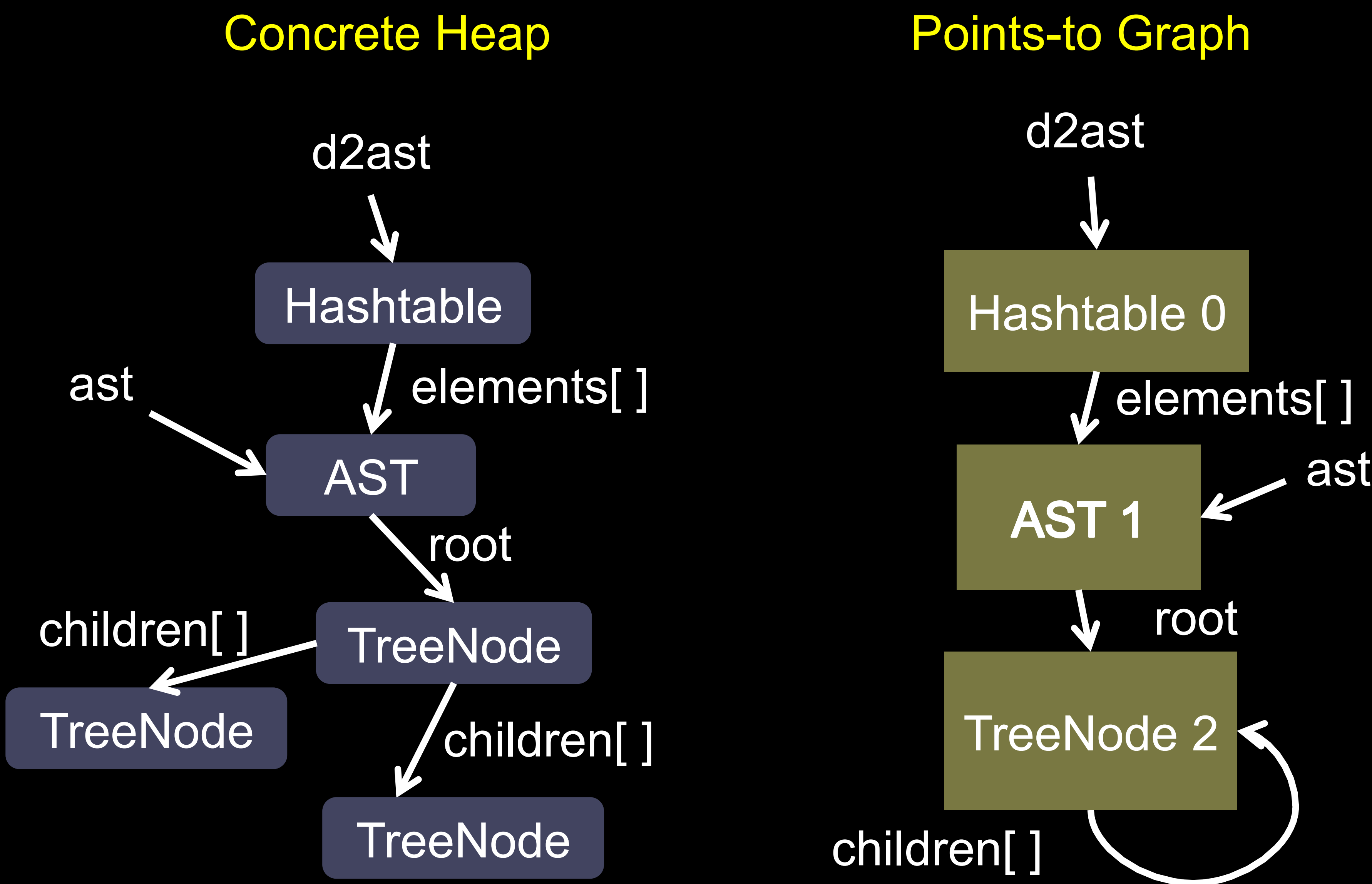
- Safe to access Node objects concurrently

An Efficient Approach

- Relate memory access to in-set variable
- Use reachability from in-set objects instead of traversing heap

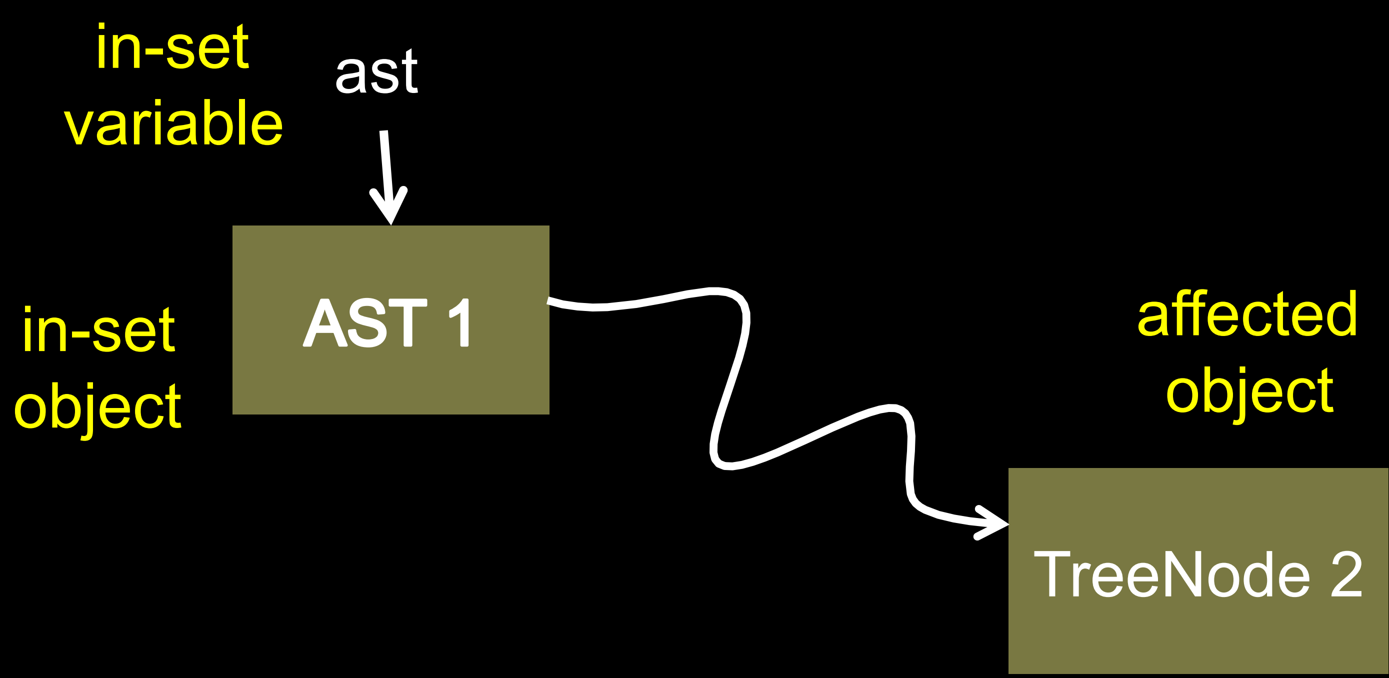


Heap Abstraction

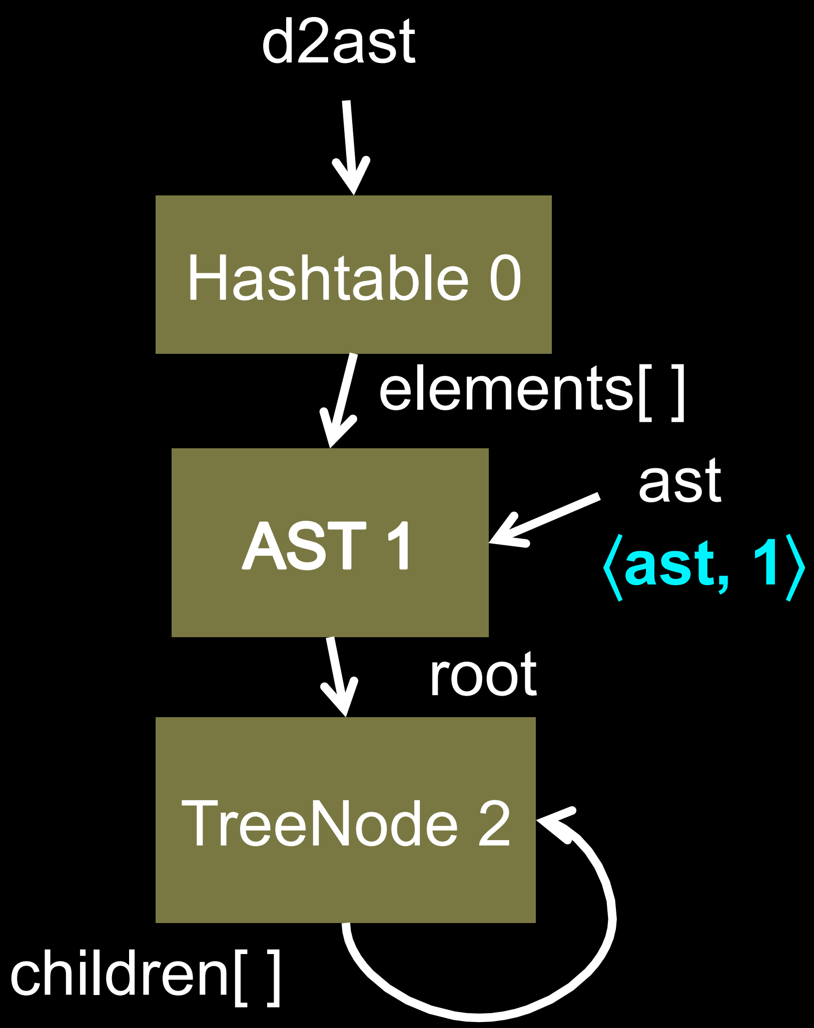


Effect Abstraction

⟨ast, 1, 2, write, f⟩ means:

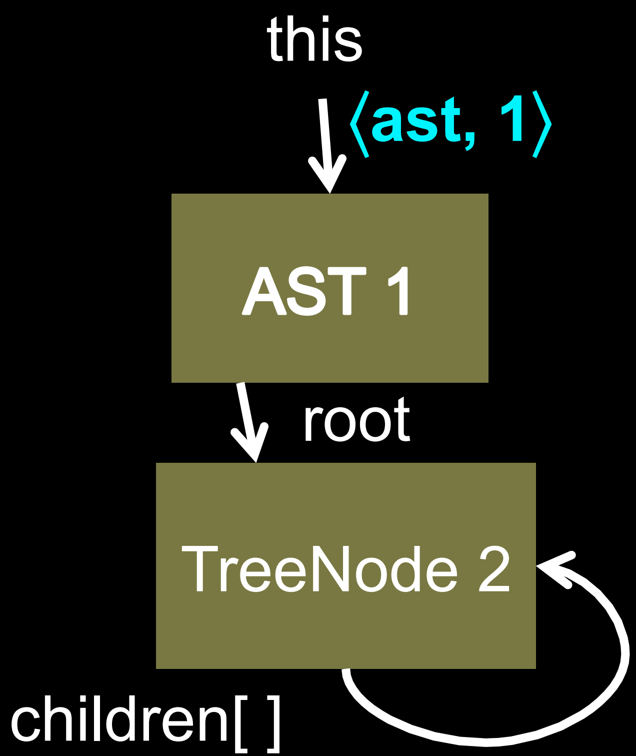


Effects Example



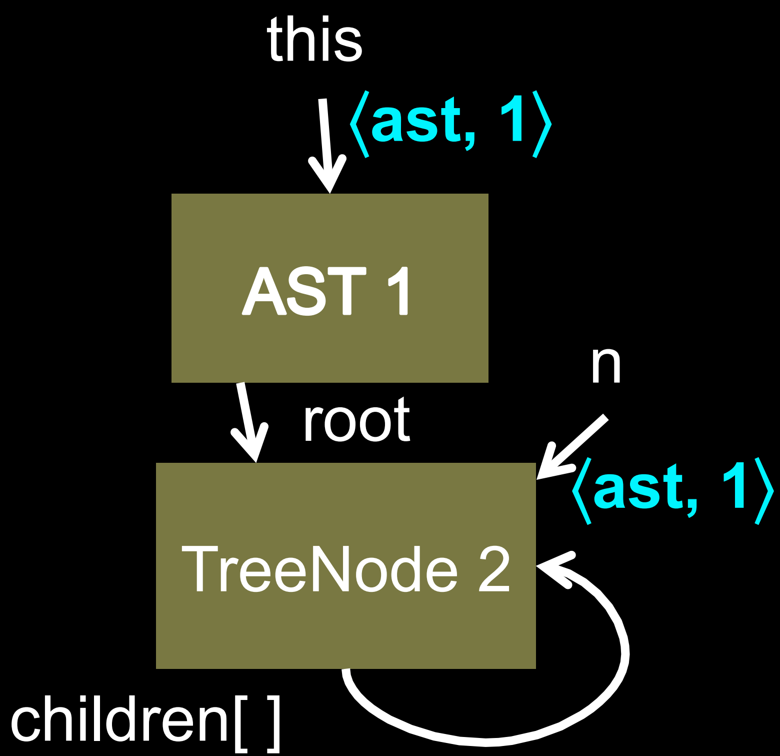

```
ast=d2ast.get(m);
rblock p {
   ast.typeCheck();
   ...
}
```

Effects Example




```
public class AST {
…
public void
typeCheck() {
   TreeNode n=this.root;
   n.type=…;
   n=n.children[i];
   …
}
```

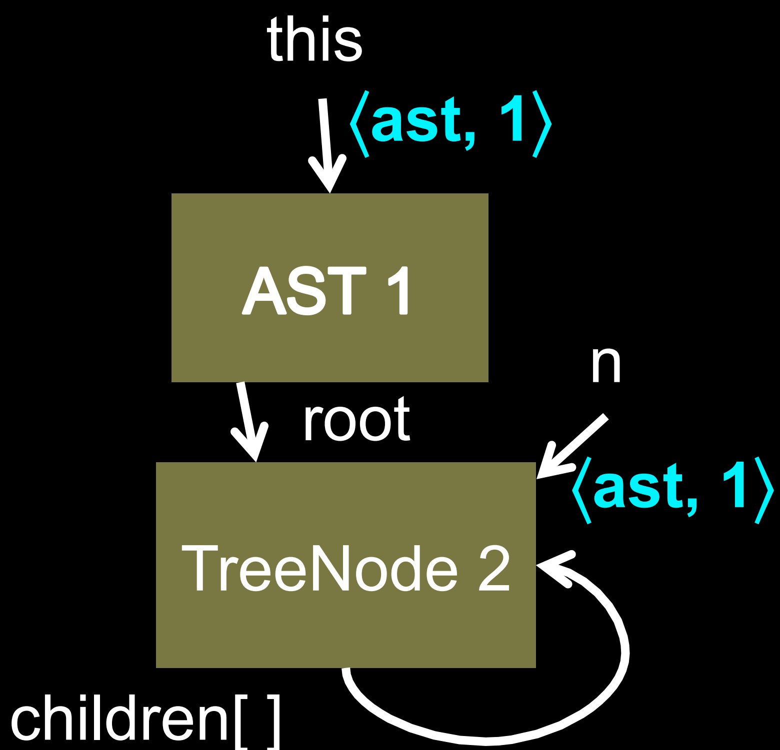
Effects Example




```
public class AST {
…
public void
typeCheck() {
    TreeNode n=this.root;
    n.type=…;
    n=n.children[i];
    …
}
```

Effects:
⟨ast, 1, 1, read, root⟩

Effects Example



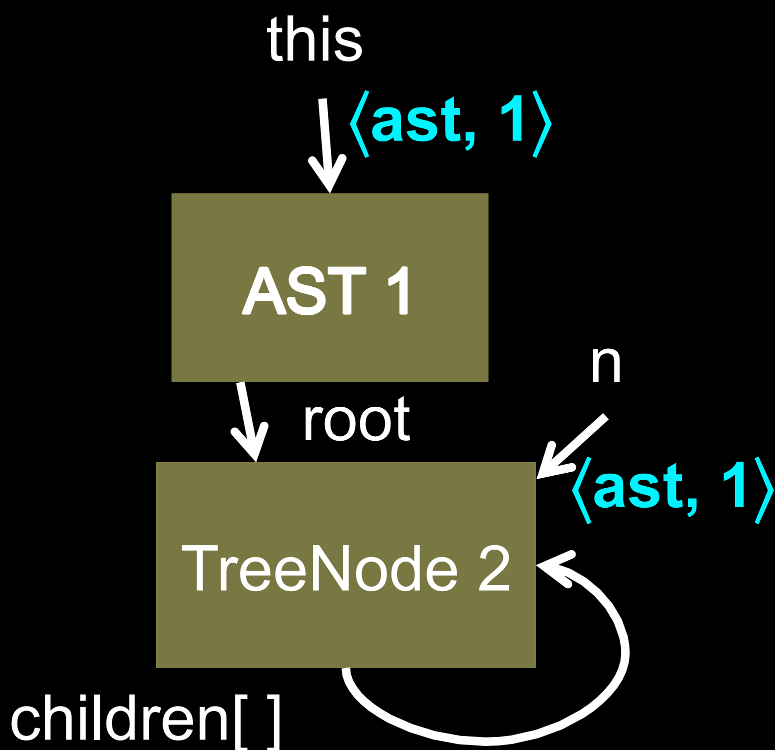

```
public class AST {
…
public void
typeCheck() {
   TreeNode n=this.root;
   n.type=…;
   n=n.children[i];
   …
}
```

Effects:

⟨ast, 1, 1, read, root⟩

⟨ast, 1, 2, write, type⟩

Effects Example



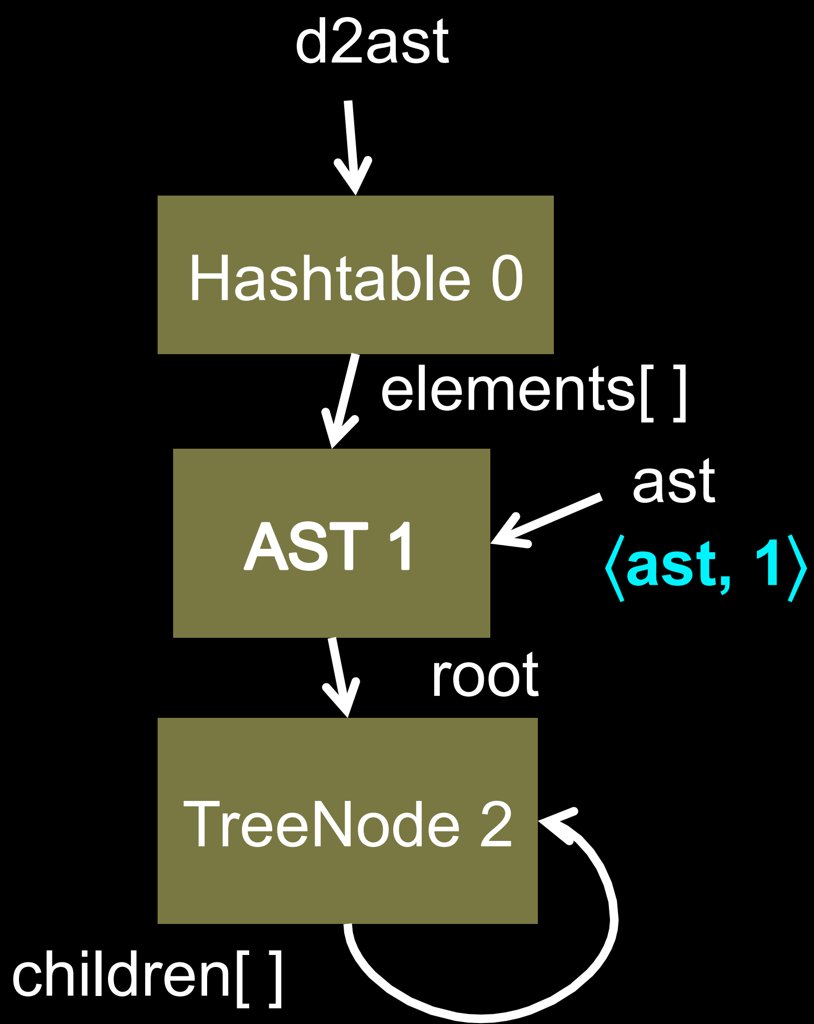

```
public class AST {
…
public void
typeCheck() {
   TreeNode n=this.root;
   n.type=…;
   n=n.children[i];
   …
}
```

Effects:

⟨ast, 1, 1, read, root⟩ ⟨ast, 1, 2, read, children⟩

⟨ast, 1, 2, write, type⟩

Effects Example




```
ast=d2ast.get(m);
rblock p {
  ast.typeCheck();
  ...
}
```

Effects for rblock p:
⟨ast, 1, 1, read, root⟩
⟨ast, 1, 2, read, children⟩
⟨ast, 1, 2, write, type⟩

Effects Example

```
ast=d2ast.get(m);
rblock p {
   ast.typeCheck();
   ...
}
```

Effects for rblock p:
⟨ast, 1, 1, read, root⟩
⟨ast, 1, 2, read, children⟩
⟨ast, 1, 2, write, type⟩

Potential conflict!

Different Object Rule

⟨ast, 1, 1, read, root⟩

Different allocation sites ⇒ No conflict

⟨ast, 1, 2, write, type⟩

Read Rule

$\langle$ast, 1, 2, read, children$\rangle$

Both accesses are reads $\Rightarrow$ No conflict

$\langle$ast, 1, 2, read, children$\rangle$

Different Field Rule

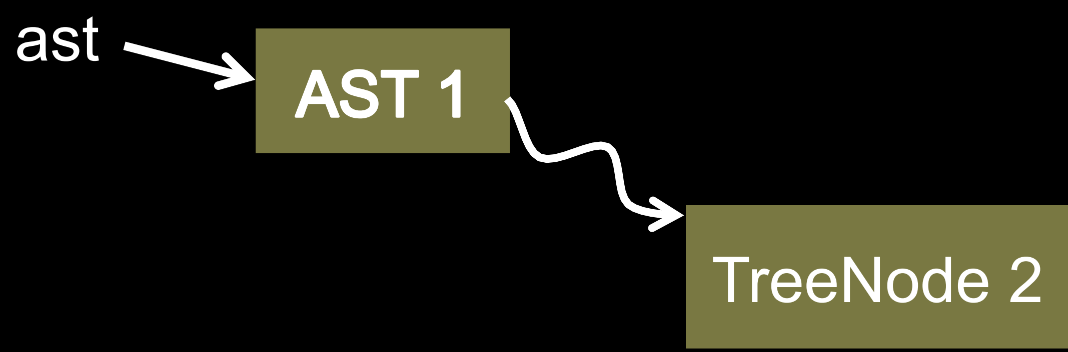
⟨ast, 1, 2, write, type⟩

Different fields ⇒ No conflict

⟨ast, 1, 2, read, children⟩

Conflict?

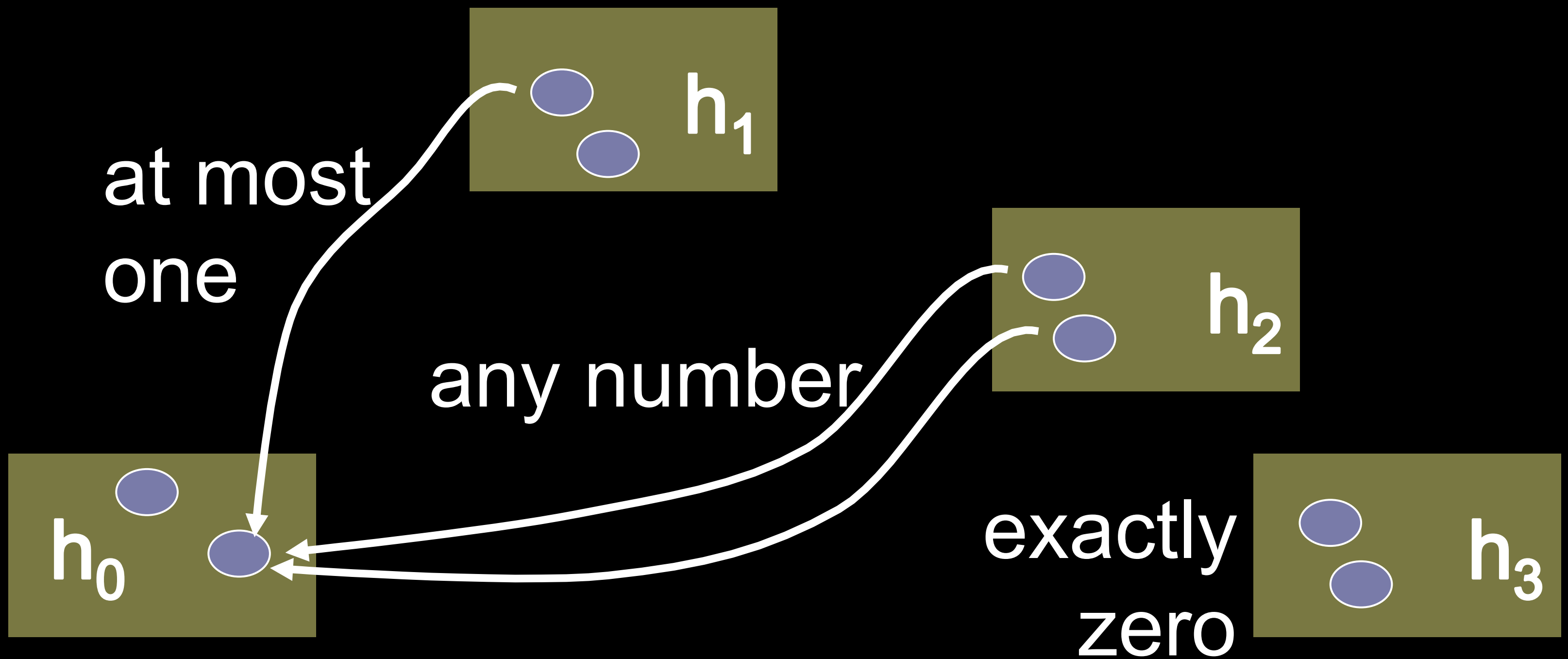
⟨ast, 1, 2, write, type⟩



⟨ast, 1, 2, write, type⟩

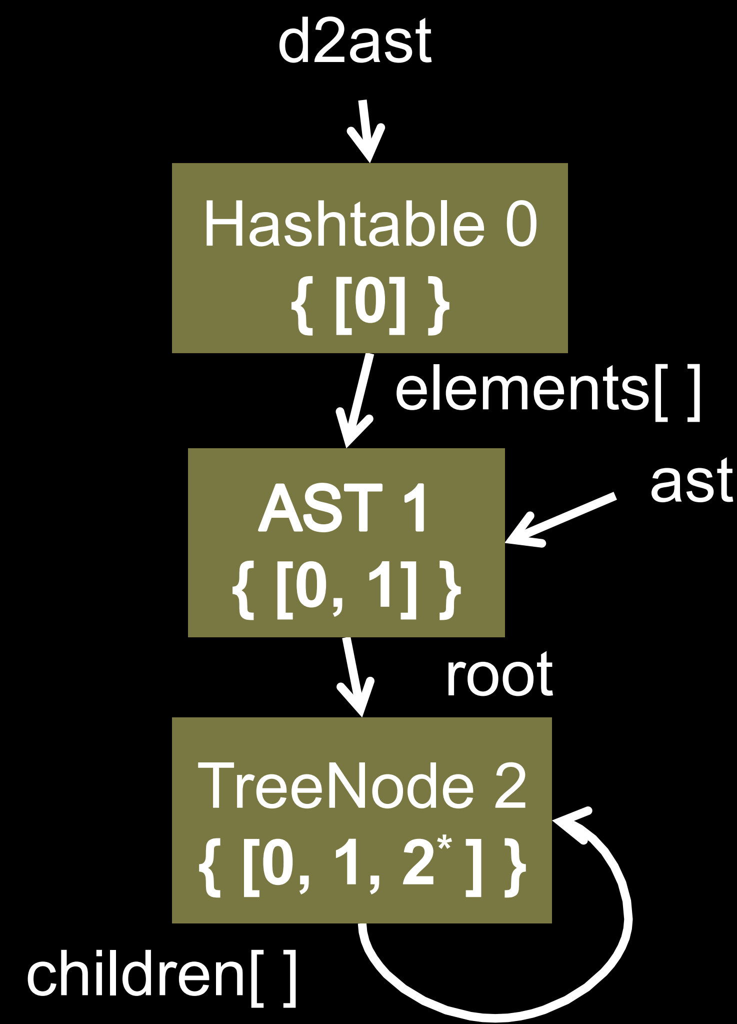
Disjoint Reachability Analysis

- Augments points-to graph with reachability states
- Region $\mathbf{h_0}$ with state $\mathbf{[\ h_1,\ h_2^{*}\ ]}$ means:



Disjoint Reachability Example

Abstraction with Reachability



Disjoint Reachability Example

Key observation:
TreeNode objects are reachable from at most one AST object

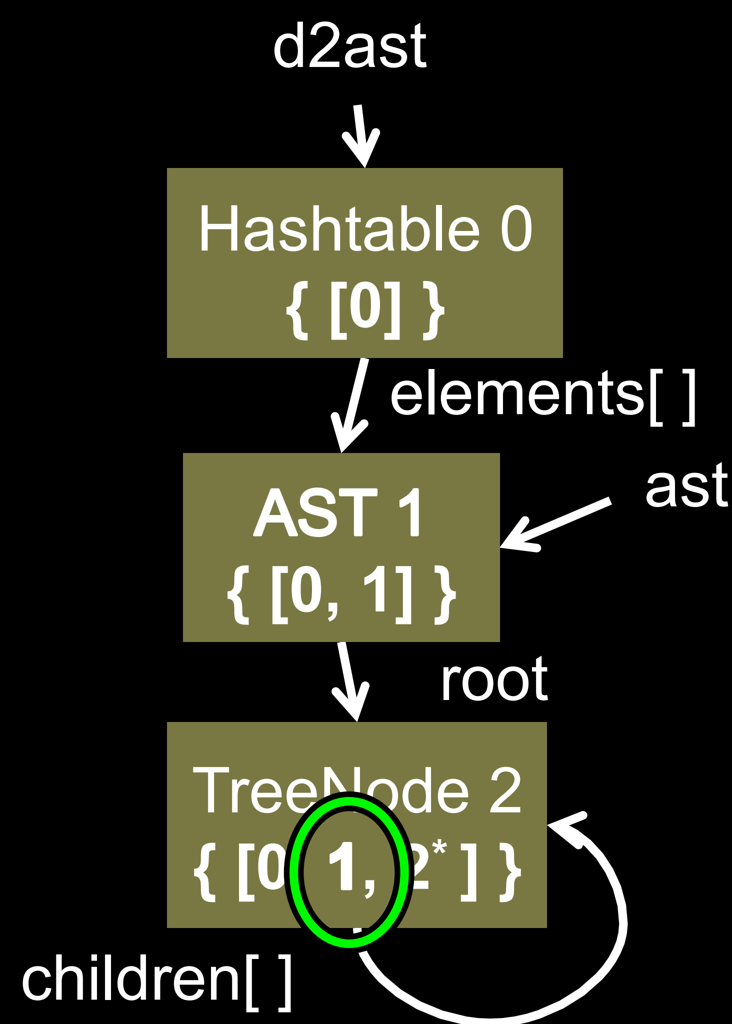
+

distinct AST in-set objects

⟹

disjoint set of TreeNodes

Abstraction with Reachability

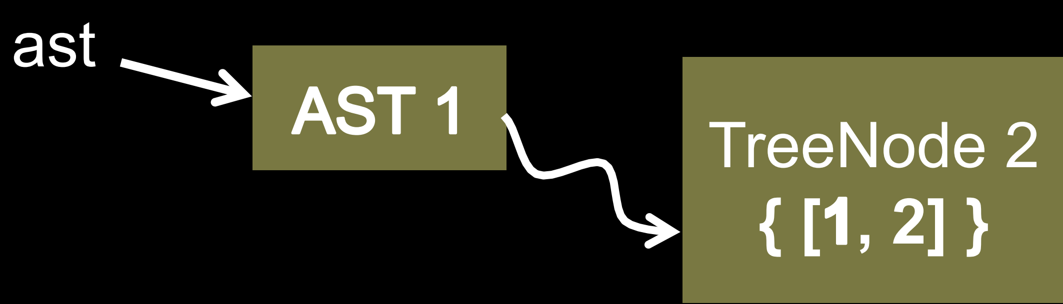


Fine-grained Conflict Rule

⟨ast, 1, 2, write, type⟩

same site + No [1^*, …] at region 2 ⇒ Dynamically check conflict

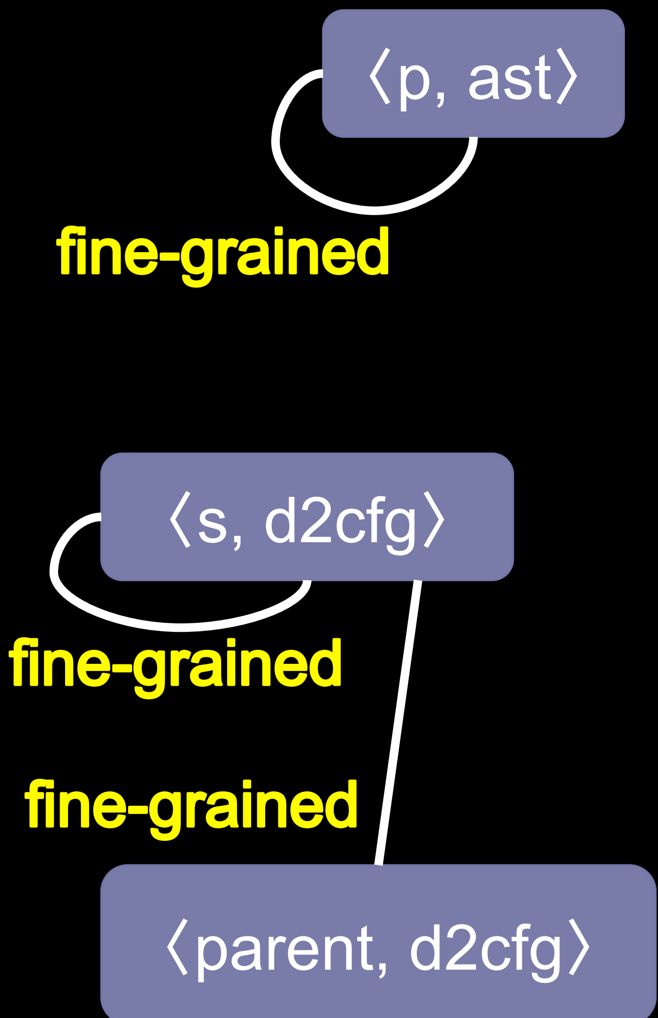
⟨ast, 1, 2, write, type⟩



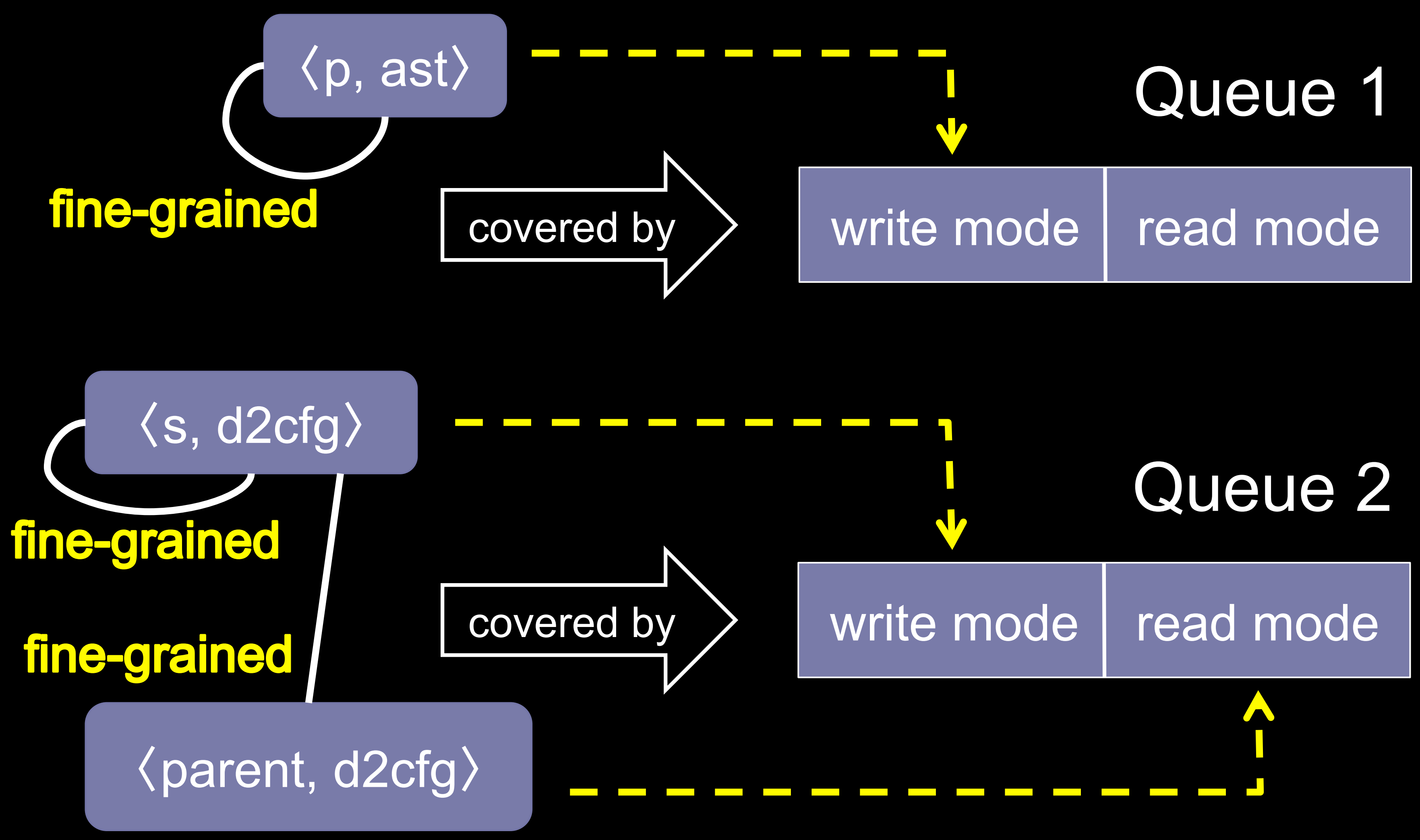
Default Rule

If no other rule eliminates a conflict, then there is a coarse-grained conflict.

Memory Conflict Graph



Compiling Conflict Graphs



Evaluation

Preliminary Evaluation

- Implemented OoOJava
- Available at http://demsky.eecs.uci.edu/compiler.php
- Executed on 2.27 GHz 8-core Intel Xeon (2 Nehalem processors)
- RayTracer – a ray tracer ported from Java Grande
- Kmeans – a data clustering algorithm ported from STAMP
- Power – power pricing algorithm ported from JOlden

Experimental Results

Benchmark	Lines of Code	Speedup
RayTracer	3,258	7.8×
Kmeans	3,541	5.8×
Power	2,275	6.0×

Related Work

- Jade (Rinard, Lam)

  OoOJava requires no access specifications
- Cilk (Frigo, Leiserson, Randall…)

  OoOJava guarantees sequential semantics and handles data structures
- Deterministic Parallel Java (Bocchino)

  OoOJava eliminates specifications
- CellSs (Perez et al)

  Strongly restricts use of data structures and variables.
- Preemptible Atomic Regions (Manson et al)

  Cannot parallelize example in this presentation

Future Work

- Evaluate the approach on a larger benchmark suite
- Extend disjoint reachability analysis to compute reachability with respect to fields accessed in rblock
- Explore I/O models (fully sequential or sequential per file)
- Explore dynamic checks to handle coarse-grained heap conflicts

Conclusion

- OoOJava preserves sequential semantics and provides strong guarantees
- Simplifies developing parallel programs
- Achieved significant speedups for our benchmarks

Questions?

Backup Slides

Variable Source Analysis

- Mapping M maps variables to a set of variable sources
- Standard set lattice definitions
  - Partial order given by $\subseteq$
  - Join given by $\cup$
  - Bottom given by $\emptyset$

copy statement: x=y

KILL = $\{x\} \times M(x)$

GEN: For each tuple $\langle r, a, v\rangle \in M(y)$

[Avoid stall case]

If r is a child of current reorderable block,

GEN includes $\langle x,\langle r,a,v\rangle\rangle$

[Have value case]

Otherwise,

GEN includes $\langle x,\langle r_{curr},0,x\rangle\rangle$

other assignments: x=expr

KILL = $\{x\} \times M(x)$

GEN: For each tuple $\langle r, a, v\rangle \in M(y)$

GEN includes $\langle x,\langle r_{curr}, 0, x\rangle\rangle$

read statement: expr(x)

[Single source case]

If $M(x) = \{\langle r', a, v_1\rangle,\ldots,\langle r', a, v_k\rangle\}$ and r' is a child of current block,

For all live variables y:

If $M(y) = \{\langle r', a, v_1'\rangle,\ldots, \langle r', a, v_l'\rangle\}$ then $\langle y, \langle r_{curr}, 0, y\rangle\rangle \in M'$

Otherwise, $M(y) \subseteq M'$

[Multiple source case]

Otherwise, if $M(x) = \{\langle r', a, v_1\rangle,\ldots\}$ and r' is a child of the current block,

For all live variables y:

If x = y, then $\langle y, \langle r_{curr}, 0, y\rangle\rangle \in M'$

Otherwise, $M(y) \subseteq M'$

[Ready case] Otherwise: M'=M

enter rblock r

For each tuple $\langle v, \langle r', a, v'\rangle\rangle \in M$

Age rblock case: r'=r

$\langle v, \langle r', a+1, v'\rangle\rangle \in M'$ if $a < k$

$\langle v, \langle r', k, v'\rangle\rangle \in M'$ otherwise

Other rblock case: $r' \neq r$

$\langle v, \langle r', a, v'\rangle\rangle \in M'$

exit rblock r

For each live variable x

If some tuples in M(x) are siblings (or older age of current rblock) and some are children or current rblock and age:

$\langle x, \langle r_{curr}, 0, x\rangle\rangle \in M'$

Ancestor or sibling source case:

For each live variable x

$\{x\} \times M(x) \subseteq M'$

Virtual Read

```
rblock a {
  x = 1;
}
```

$x \rightarrow \{ \langle a, 0, x \rangle \}$

```
rblock b {
  if( … ) {  x = 2 }
```

$x \rightarrow \{ \langle a, 0, x \rangle, \langle b, 0, x \rangle \}$

```
}
```

$x \rightarrow \{ \langle b, 0, x \rangle \}$

For each live variable x:

If sources are mix of case 1 & 2,

treat as a virtual read, force source

Virtual Reads

- Problem: rblock conditionally writes to a variable, statically difficult or impossible to decide variable's source beyond
- Solution: analysis treats this as a virtual read and adds to the variable to the rblock's in-set. Analysis forces this rblock to become the source of the variable whether it dynamically writes to the variable or not

Effects Analysis

Map L from variables to a set of in-set allocation sites (and variables)

Map R from heap edges in points-to graph to a set of in-set allocation sites (and variables).

Standard Set Lattice

copy statement: x=y

KILL = {x} × L(x)

GEN = {x} x L(y)

store statement: x.f=y

L' := L

R' := R ∪ E(x,f) x L(y)

load statement: x=y.f

KILL = {x} × L(x)

GEN = {x} x L(y) ∪ {x} x R(E(y, f))

L' := (L – KILL) ∪ GEN

R' := R

method calls

Details depend on points-to analysis method call abstraction.

Parent effects

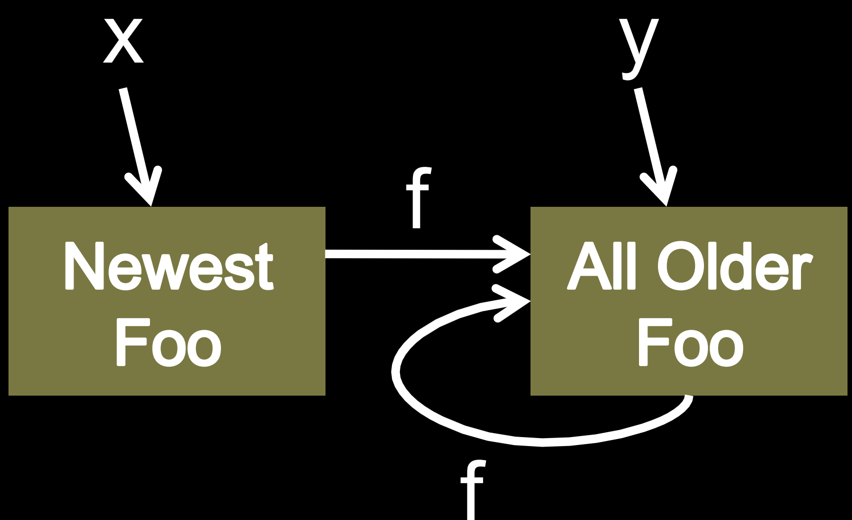
Must generate effects for regions of parent rblock after any child rblocks

- Do same analysis using live variables into this region (instead of in-set variables)

Disjoint Reachability Analysis

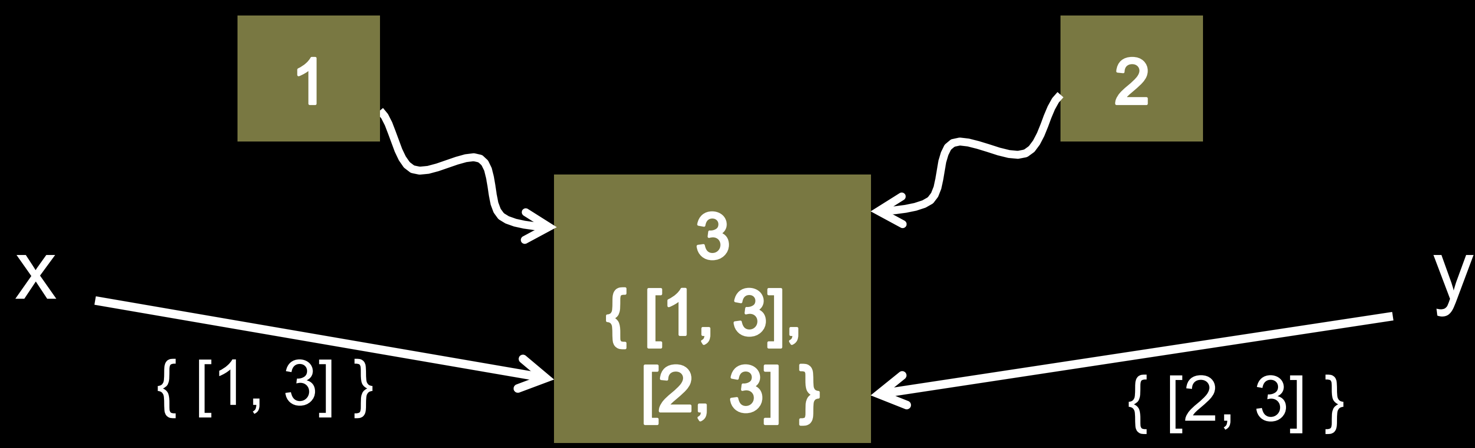
- Abstract objects by allocation site:
  - single-object region for most recent object
  - multi-object region for all older objects

```
while( … )
{
  y = x;
  x = new Foo();
  x.f = y;
}
```



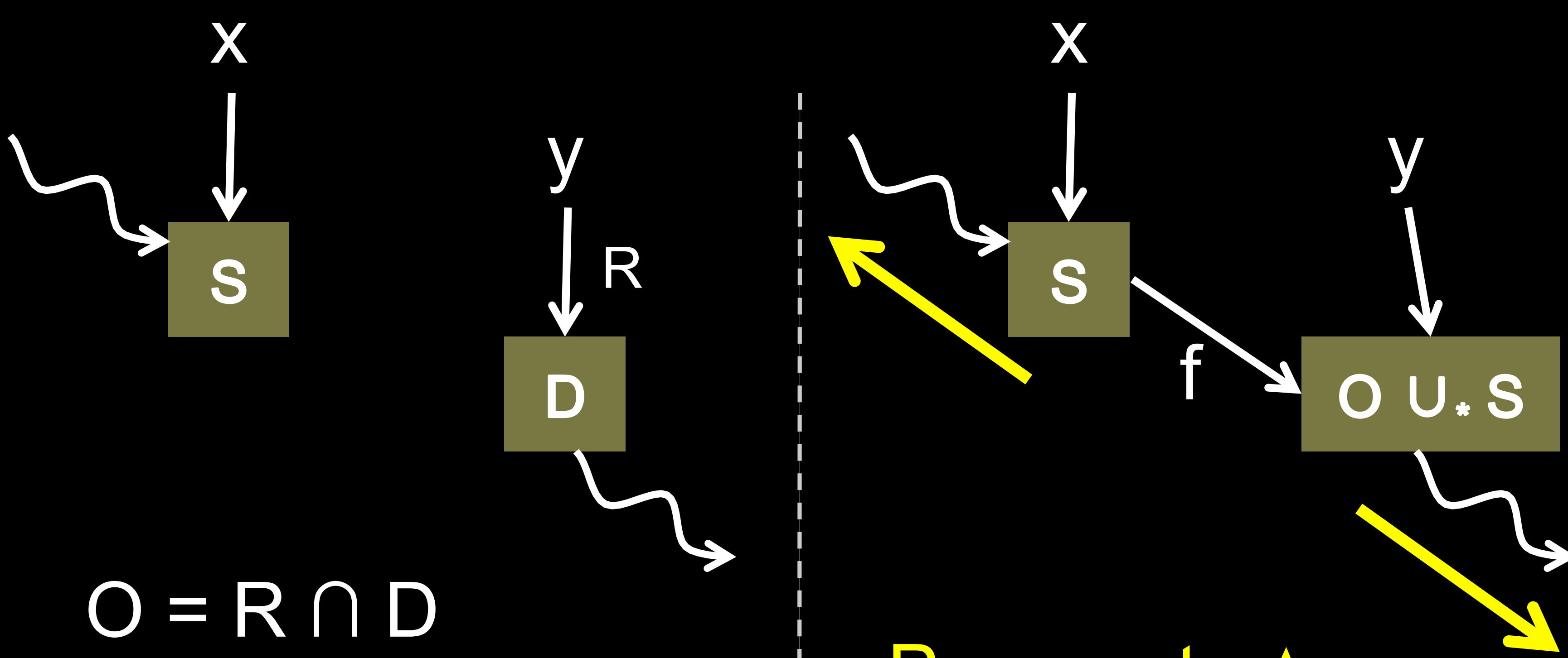
Disjoint Reachability Analysis

- Reach state on node: object represented by this node is reachable from objects of regions in state
- Reach state on edge: an object with this reach state is reachable through this reference



Disjoint Reachability Analysis

- Key reachability transfer function: x.f = y



$O = R \cap D$

$\Delta = O \rightarrow O \cup_{*} S$

Propagate Δ

Reachability Conflict Condition

Effects of two rblocks only conflict when

(1) conflict still possible after previous rules and

(2) affected objects are reachable from the in-set object of both effects*.

* If the reachability of the affected objects does not decrease.

STRONG UPDATES DO THIS!

Reachability Rule

⟨ast, 1, 2, write, children⟩

⟨astPrime, 7, 2, write, children⟩

No strong update + No [1, 7, …] at region 2 $\Rightarrow$ No conflict