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

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

    d2cfg.put( m, cfg );
}

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

d2cfg.serializeToDisk();
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();
Out-of-Order Java Execution

 Issued

 Ready

 Executing

 Core

 Core

 Core

 Core

 Stalled

 Retired

 parent
Out-of-Order Java Execution

- Issued
  - Ready
    - parent
  - Executing
    - Core
    - Core
    - Core
    - Core
  - Stalled
- Retired
while(...)
{
    Descriptor m= ...;
    ast=d2ast.get(m);
    rblock p {
        ast.typeCheck();
        cfg=ast.flatten();
    }
    rblock s {
        d2cfg.put(m, cfg);
    }
}
d2cfg.serializeToDisk();
while(...) {
    Descriptor m= ...;
    ast=d2ast.get( m );
    rblock p {
        ast.typeCheck();
        cfg=ast.flatten();
    }
    rblock s {
        d2cfg.put( m, cfg );
    }
}
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 );
    }
}
Out-of-Order Java Execution

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

d2cfg.serializeToDisk();
Out-of-Order Java Execution

```java
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 );
    }
}
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

```java
while(...) {
  Descriptor m = ...;
  ast = d2ast.get(m);
  rblock p {
    ast.typeCheck();
    cfg = ast.flatten();
  }
  rblock s {
    d2cfg.put(m, cfg);
  }
}
```
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);
    }
}
Out-of-Order Java Execution

```java
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

```java
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

```java
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
  
  \[
  \text{rblock a} \{ \\
  \ldots \\
  \text{rblock b} \{ \\
  \} \\
  \}
  \]

- 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
Many instances of an rbblock 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

```java
rblock p {
    ast.typeCheck();
    ```

    ```
    cfg = ast.flatten();
    ```
}

rblock s {
    ```
    d2cfg.put( m, cfg );
    ```
}
```
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 → { ⟨c, 0, x⟩ }

  Kill facts for x

  x = w.g;

  x → { ⟨r, 0, x⟩ }

}
Reading Variables

```
rblock r {  
rblock c { x = v.f; }  
  x → { ⟨c, 0, x⟩ }  

  Kill facts for x  

  ... = x.g;  
  x → { ⟨r, 0, x⟩ }  
}
```
Avoiding Stalls

```c
rblock r {
    rblock c { y = v.f; }
    y → { ⟨c, 0, y⟩ }
    Kill facts for x
    x = y;
    x → { ⟨c, 0, y⟩ }, y → { ⟨c, 0, y⟩ }
}
```
rblock enter

\[
x \rightarrow \{ \langle r, 0, x \rangle \}
\]

\[
y \rightarrow \{ \langle q, 0, y \rangle \}
\]

\[
rblock r \{ \\
x \rightarrow \{ \langle r, 1, x \rangle \} \\
y \rightarrow \{ \langle q, 0, y \rangle \}
\}
\]
A parent cannot retire until all its children have retired

Bad for humans, good for rblocks!
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 → { ⟨c, 0, x⟩ }

}

x → { ⟨r, 0, x⟩ }
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
- 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.
To issue rblock p, for each previous rblock q that has not retired:
A Brute Force Approach

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

  TOO EXPENSIVE!
Solution: Use Reachability

- Every Node in heap is reachable from at most one Graph object
  - p’s g1 ≠ q’s g2
  - 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

Concrete Heap

Points-to Graph

```
# Graph Representation

d2ast

Hashtable

ast

elements[

AST

root

TreeNode

children[

TreeNode

ast

elements[

TreeNode

AST 1

root

TreeNode 2

children[
```
Effect Abstraction

\[ \langle \text{ast}, 1, 2, \text{write, f} \rangle \] means:

in-set variable

in-set object

ast

AST 1

affected object

TreeNode 2
Effects Example

```
ast = d2ast.get(m);

rblock p {
    ast.typeCheck();
    ...
}
```
Effects Example

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

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

**Effects:**
\langle \text{ast, 1, 1, read, root} \rangle
Effects Example

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

Effects:

\(<\text{ast, 1, 1, read, root}\>\)
\(<\text{ast, 1, 2, write, type}\>\)
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

```java
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

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

Potential conflict!

Effects for `rblock p`:
- `<ast, 1, 1, read, root>`
- `<ast, 1, 2, read, children>`
- `<ast, 1, 2, write, type>`
Different Object Rule

\[ \langle \text{ast}, 1, 1, \text{read, root} \rangle \]

\[ \downarrow \]

Different allocation sites

\[ \Rightarrow \]

No conflict

\[ \langle \text{ast}, 1, 2, \text{write, type} \rangle \]
Read Rule

\[ \langle \text{ast, 1, 2, read, children} \rangle \]

Both accesses are reads \implies \text{No conflict}

\[ \langle \text{ast, 1, 2, read, children} \rangle \]
Different Field Rule

\[ \langle \text{ast, 1, 2, write, type} \rangle \]

Different fields \[\Rightarrow\] No conflict

\[ \langle \text{ast, 1, 2, read, children} \rangle \]
Conflict?

\[ \langle \text{ast, 1, 2, write, type} \rangle \]

\[ \langle \text{ast, 1, 2, write, type} \rangle \]
Disjoint Reachability Analysis

- Augments points-to graph with reachability states
- Region $h_0$ with state $[h_1, h_2^*]$ means:
  - at most one
  - any number
  - exactly zero
Disjoint Reachability Example

```
Abstraction with Reachability

d2ast

Hashtable 0
{ [0] }

elements[ ]

AST 1
{ [0, 1] }

root

TreeNode 2
{ [0, 1, 2*] }

children[ ]
```
Disjoint Reachability Example

Key observation: 
TreeNode objects are reachable from at most one AST object
+ distinct AST in-set objects
⇒ disjoint set of TreeNode objects
Fine-grained Conflict Rule

\[ \langle \text{ast, 1, 2, write, type} \rangle \]

same site + No \([1^*, \ldots]\) at region 2 \(\Rightarrow\) Dynamically check conflict

\[ \langle \text{ast, 1, 2, write, type} \rangle \]

\(\text{ast} \rightarrow \text{AST 1} \rightarrow \text{TreeNode 2} \{[1, 2]\} \)
Default Rule

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

- ⟨p, ast⟩
  - fine-grained

- ⟨s, d2cfg⟩
  - fine-grained
  - fine-grained

- ⟨parent, d2cfg⟩
Compiling Conflict Graphs

fine-grained

\(\langle p, \text{ast} \rangle\)

\(\langle s, \text{d2cfg} \rangle\)

\(\langle \text{parent, d2cfg} \rangle\)

Queue 1

write mode  read mode

Queue 2

write mode  read mode
Evaluation
Preliminary Evaluation

- Implemented OoOJava
- Available at http://demsyke.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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>RayTracer</td>
<td>3,258</td>
<td>7.8×</td>
</tr>
<tr>
<td>Kmeans</td>
<td>3,541</td>
<td>5.8×</td>
</tr>
<tr>
<td>Power</td>
<td>2,275</td>
<td>6.0×</td>
</tr>
</tbody>
</table>
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: $x=y$**

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

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

- **[Avoid stall case]**
  - If $r$ is a child of current reorderable block,
    - $\text{GEN}$ includes $\langle x, \langle r, a, v \rangle \rangle$

- **[Have value case]**
  - Otherwise,
  - $\text{GEN}$ includes $\langle x, \langle r_{\text{curr}}, 0, x \rangle \rangle$

**other assignments: $x=\text{expr}$**

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

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

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

**read statement: $\text{expr}(x)$**

- **[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 $x = y$, then $\langle y, \langle r_{\text{curr}}, 0, y \rangle \rangle \in M'$
    - Otherwise, $M(y) \subseteq M'$

- **[Multiple source case]**
  - Otherwise, if $M(x) = \{\langle r', a, v_1 \rangle, \ldots, \langle r', a, v_k \rangle\}$ and $r'$ is a child of the current block,
  - For all live variables $y$:
    - If $x = y$, then $\langle y, \langle r_{\text{curr}}, 0, y \rangle \rangle \in M'$
    - 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 rbblock) and some are children or current rbblock and age:
  - $\langle x, \langle r_{\text{curr}}, 0, x \rangle \rangle \in M'$

- Ancestor or sibling source case:
  - For each live variable $x$
    - $\{x\} \times M(x) \subseteq M'$
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\} \times L(x)
GEN = \{x\} \times L(y)

**load statement: x=y.f**
KILL = \{x\} \times L(x)
GEN = \{x\} \times L(y) \cup \{x\} \times R(E(y, f))
L’ := (L – KILL) \cup GEN
R’ := R

**store statement: x.f=y**
L’ := L
R’ := R \cup E(x,f) \times L(y)

**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

```java
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 $\Delta$
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

\[ \langle \text{ast}, 1, 2, \text{write, children} \rangle \]
\[ \langle \text{astPrime}, 7, 2, \text{write, children} \rangle \]

No strong update + No \([1, 7, \ldots]\) at region 2 \implies\ No conflict

```
TreeNode 2
{ [ 1, 2 ],
  [ 7, 2 ] }
```

AST 1

AST 7

TreeNode 2