Fast and precise static race detection for loop-parallelism in Java programs

Cosmin Rădoi (joint work with Danny Dig)
Why not dynamic race detectors?

existing dynamic analyses...
  ... require running the program
  => usually have runtime overhead
  => input dependent

=> do not catch all races
Why not an existing static race detector?

existing static analyses...
  ... improved a lot in the last decade
  but,
    - still do not scale or
    - report many false races

=> less used than dynamic analysis
Can better results be achieved by focusing on specific parallelism pattern, e.g. loop parallelism?

Yes.

This allows cheaper and more precise solutions for aliasing, lock identity and array sensitivity.
Why loop parallelism?

common practice for speeding up software

well supported by parallel frameworks

- C++ TBB: parallel_for
- .NET TPL: Parallel.For
- Java: ParallelArray
Why race detection?

non-determinism

=> concurrency bugs, including races, are hard to find

OO programs frequently share data through the heap

=> races are common
Motivating example

```java
particles.generate(new Generator<Particle>() {
    public Particle op() {
        Particle p = new Particle();
        readParticle(p);
        return p;
    }
});
for (int i = 0; i < noSteps; i++) {
    // ... update force ...

    particles.apply(new Procedure<Particle>() {
        public void op(Particle p) {
            move(p);
        }
    });
    particles.apply(new Procedure<Particle>() {
        public void op(Particle p) {
            computeCenterOfMass();
        }
    });
}
```
Motivating example

```java
public Particle op() {
    Particle p = new Particle();
    readParticle(p);
    return p;
}
```
Motivating example

```java
particles.generate(...
    public Particle op() {
        Particle p = new Particle();
        readParticle(p);
        return p;
    }
});
for (int i = 0; i < noSteps; i++) {
    // ... update force ...
}

particles.apply(...
    public void op(Particle p) {
        move(p)
    }
});
particles.apply(...
    public void op(Particle p) {
        computeCenterOfMass();
    }
});
```
Motivating example

```java
particles.generate(...)  
    public Particle op() {  
        Particle p = new Particle();  
        readParticle(p);  
        return p;  
    }  
});

[...]  

particles.apply(...)  
    public void op(Particle p) {  
        move(p)  
    }  
});
```
Motivating example

```java
particles.generate(...
    public Particle op() {
        Particle p = new Particle();
        readParticle(p);
        return p;
    }
});

[...]

particles.apply(...
    public void op(Particle p) {
        p.vx += p.fx / p.m * dt;
        p.vy += p.fy / p.m * dt;
        p.x += p.vx * dt;
        p.y += p.vy * dt;
    }
});
```
Previous approaches reason about one thread

particles.generate(...
    public Particle op() {
        Particle p = new Particle();
        readParticle(p);
        return p;
    }
}));

[...]

particles.apply(...
    public void op(Particle p) {
        p.vX += p.fX / p.m * dT;
        p.vY += p.fY / p.m * dT;
        p.x += p.vX * dT;
        p.y += p.vY * dT;
    }
}));
Previous approaches reason about one thread

```java
public Particle op() {
    Particle p = new Particle();
    readParticle(p);
    return p;
}
}

public void op(Particle p) {
    p.vX += p.fX / p.m * dT;
    p.vY += p.fY / p.m * dT;
    p.x += p.vX * dT;
    p.y += p.vY * dT;
}
```
Our approach reasons about two threads

```java
particles.generate(...) {
    public Particle op() {
        Particle p1 = new Particle();
        readParticle(p1);
        return p1;
    }
}

// T1

[...]

particles.apply(...) {
    public void op(Particle p1) {
        p1.vX += p1.fX / p1.m * dT;
        p1.vY += p1.fY / p1.m * dT;
        p1.x += p1.vX * dT;
        p1.y += p1.vY * dT;
    }
}

// T1

particles.generate(...) {
    public Particle op() {
        Particle p2 = new Particle();
        readParticle(p2);
        return p2;
    }
}

// T2

[...]

particles.apply(...) {
    public void op(Particle p2) {
        p2.vX += p2.fX / p2.m * dT;
        p2.vY += p2.fY / p2.m * dT;
        p2.x += p2.vX * dT;
        p2.y += p2.vY * dT;
    }
}

// T2
```
Our approach reasons about two threads

```
particles.generate(...

public Particle op() {
    Particle p1 = new Particle();
    readParticle(p1);
    return p1;
}

});

[...]  

particles.apply(...

public void op(Particle p1) {
    p1.vX += p1.fX / p1.m * dT;
    p1.vY += p1.fY / p1.m * dT;
    p1.x += p1.vX * dT;
    p1.y += p1.vY * dT;
}

});

T1
```

```
particles.generate(...

public Particle op() {
    Particle p2 = new Particle();
    readParticle(p2);
    return p2;
}

});

[...]  

particles.apply(...

public void op(Particle p2) {
    p2.vX += p2.fX / p2.m * dT;
    p2.vY += p2.fY / p2.m * dT;
    p2.x += p2.vX * dT;
    p2.y += p2.vY * dT;
}

});

T2
```
Motivating example

```java
particles.generate(...
    public Particle op() {
        Particle p = new Particle();
        readParticle(p);
        return p;
    }
});
for (int i = 0; i < noSteps; i++) {
    // ... update force ...

    particles.apply(...
        public void op(Particle p) {
            move(p)
        }
    });
}
particles.apply(...
    public void op(Particle p) {
        computeCenterOfMass();
    }
});
```
Race when computing center of mass

```java
class Computation {
    // center of mass
    Particle c = new Particle();

    void compute() {
        [...]
    }
}
```

```java
particles.apply(...) {
    public void op(Particle p1) {
        Particle oldC = c;
        this.c = new Particle();
        c.m = oldC.m + p.m;
        c.x = (oldC.x * ...;
        c.y = (oldC.y * ...;
    }
}
```

```java
particles.apply(...) {
    public void op(Particle p1) {
        Particle oldC = c;
        this.c = new Particle();
        c.m = oldC.m + p.m;
        c.x = (oldC.x * ...;
        c.y = (oldC.y * ...;
    }
}
```
Race when computing center of mass

```java
class Computation {
    // center of mass
    Particle c = new Particle();
    void compute() {
        [...]}
}
particles.apply(...)
public void op(Particle p1) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p1.m;
    c.x = (oldC.x * ...  
    c.y = (oldC.y * ...
});//

T1
particles.apply(...)
public void op(Particle p2) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p2.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
});//

T2
```
Race when computing center of mass

```java
public void op(Particle p1) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p1.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
}
```

```java
public void op(Particle p2) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p2.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
}
```

particles.apply(...)

T1

T2
Race when computing center of mass

```
public void op(Particle p1) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p1.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
}
```

```
public void op(Particle p2) {
    Particle oldC = c;
    this.c = new Particle();
    c.m = oldC.m + p2.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
```
System overview

- Shallow races
  - Bubble out of JDK
    - Lock identity analysis
      - Lock sets
        - Data-flow analyses
          - CG - CFGs
      - Concurrent instructions
        - Traverse and match
          - Heap Graph
    - Concurrency instructions
      - Traverse and match
        - Heap Graph
  - Pointer analysis
    - CG - CFGs
Pointer analysis

Andersen style analysis

special context sensitivity:

- 0-CFA to 1-CFA outside of loops, depending on the number of arrays the tool reasons about
- array object and loop iteration (our modeling)
- 1-object sensitivity on edge of JDK
System overview

- Shallow races
- Bubble out of JDK
- Races
- Lock identity analysis
- Concurrent instructions
- Traverse and match
- Lock sets
- Data-flow analyses
- CG - CFGs
- Heap Graph
- Pointer analysis
Concurrent Instructions

1. find all reads and writes in the parallel loops

2. match them based on aliases

=> pairs of instructions that can race
Lock sets

Inter-procedural data-flow analysis is needed
Lock sets analysis

Intra-procedural data-flow analysis
- over each method's CFG
- the lattice is the lock set power set

Inter-procedural data-flow analysis
- over call graph
- transfer functions on edges
 (call site to call graph node)
System overview

- shallow races
- bubble out of JDK
- races
- lock identity analysis
- lock sets
- concurrent instructions
- data-flow analyses
- traverse and match
- CG - CFGs
- Heap Graph
- pointer analysis
Lock identity. Why?

```
class Computation {
  // center of mass
  Particle c = new Particle();
  Object lock;
  void compute() {
    [..]
  }
}
	particles.apply(...

public void op(Particle p1) {
  Particle oldC = c;
  synchronized(lock) {
    this.c = new Particle();
    c.m = oldC.m + p.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
  }

}());
```

```
public void op(Particle p1) {
  Particle oldC = c;
  synchronized(lock) {
    this.c = new Particle();
    c.m = oldC.m + p.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
  }

}());
```
Lock identity. Why?

class Computation {
    // center of mass
    Particle c = new Particle();
    Object lock;
    void compute() {
        [...]
    }
}

class Computation {
    // center of mass
    Particle c = new Particle();
    Object lock;
    void compute() {
        [...]
    }
}

public void op(Particle p1) {
    Particle oldC = c;
    synchronized(lock) {
        this.c = new Particle();
    }
    c.m = oldC.m + p.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
}

public void op(Particle p1) {
    Particle oldC = c;
    synchronized(lock) {
        this.c = new Particle();
    }
    c.m = oldC.m + p.m;
    c.x = (oldC.x * ...
    c.y = (oldC.y * ...
}

particles.apply(...)
particles.apply(...)
Lock identity

may alias not enough

must alias needed
Lock identity

Conditional must-alias:

- represent each lock variable in a call graph node as a unique list of field dereferences from a reference that is know as unique, e.g. the array reference or a static field
Lock identity: Why?

```java
class Computation {
   // center of mass
   Particle c = new Particle();
   Object lock;
   void compute() {
      [...]}
}
particles.apply(...)
```
System overview

- shallow races
- bubble out of JDK
- races
- lock identity analysis
- lock sets
- concurrent instructions
- traverse and match
- data-flow analyses
- pointer analysis
- CG - CFGs
- Heap Graph
Motivating example

```java
class Computation {
    HashSet results = new HashSet();

    void compute() {
        [...]
    }

    particles.apply(...
        public void op(Particle p1) {
            results.add(p1);
        }
    });

    particles.apply(...
        public void op(Particle p1) {
            results.add(p1);
        }
    });

    ... in HashMap ...
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    ...
Sources of unsoundness

classic, reflection and native code

other threads, aside from ParallelArray

... their existence can be easily checked

ParallelArray.getArray - shuffling the elements of the array outside the parallel array operators

... can easily be checked
Evaluation: Research questions

Is the analysis effective?
  - total number of reported races
  - false races
  - missed races

Is the analysis efficient?
  - runtime of the analysis
Evaluation methodology - subjects

select programs that use ParallelArray

... still few to none
Evaluation methodology - subjects

1. select programs with algorithms that have potential for loop parallelism

2. parallelize them using ParallelArray
Evaluation methodology - compare

state of the art static race detection for java - JChord

Effective Static Race Detection for Java. Mayur Naik, Alex Aiken, and John Whaley.
PLDI'06: ACM Conference on Programming Language Design and Implementation

Conditional Must Not Aliasing for Static Race Detection. Mayur Naik and Alex Aiken.
POPL '07: ACM Symposium on Principles of Programming Languages
# Evaluation results

<table>
<thead>
<tr>
<th>Project name</th>
<th>Size (SLOC)</th>
<th>JDK (filtered)</th>
<th>Bugs</th>
<th>Analysis time (s)</th>
<th>JChord Races</th>
<th>Missed bugs</th>
<th>Analyzed invocations</th>
<th>Analysis time (s)</th>
<th>ReLooper Races</th>
<th>Missed bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes-Hut sim.</td>
<td>899</td>
<td>220k</td>
<td>0</td>
<td>3m:05s</td>
<td>108</td>
<td>0</td>
<td>3197</td>
<td>20s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EM wave prop.</td>
<td>181</td>
<td>220k</td>
<td>0</td>
<td>1m:56s</td>
<td>16</td>
<td>0</td>
<td>2339</td>
<td>14s</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MonteCarlo sim.</td>
<td>1441</td>
<td>220k</td>
<td>2</td>
<td>3m:10s</td>
<td>249</td>
<td>0</td>
<td>1473</td>
<td>12s</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Coref</td>
<td>41k</td>
<td>220k</td>
<td>2</td>
<td>8m:59s</td>
<td>4622</td>
<td>2</td>
<td>11556</td>
<td>43s</td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>
Conclusion

static race detection is useful

but a general one either:
   - reports many false positives, or
   - doesn't scale

our analysis is specialized for loop parallelism
   => better results