Exploiting Task-Order Information

for Optimizing Sequentially Consistent Java Programs

Christoph M. Angerer
Thomas R. Gross
ETH Zurich, Switzerland

Sequential Consistency

- Easy-to-understand memory model
- Conceptually:
  - All memory accesses are visible immediately
  - All tasks agree on same legal sequential history of memory events
  - No re-ordering

Sequential Consistency

- Easy-to-understand memory model
- Conceptually:
  - All memory accesses are visible immediately
  - All tasks agree on same legal sequential history of memory events
  - No re-ordering

`Inefficient without optimizations!`

SC Example

Initially, x == y == 0

1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;

Is r1 == 1 AND r2 == 2 possible?

SC Example

Initially, x == y == 0

1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;

Is r1 == 1 AND r2 == 2 possible?

SC Example

Initially, x == y == 0

1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;

Is r1 == 1 AND r2 == 2 possible?

SC Example

Initially, x == y == 0

1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;

Is r1 == 1 AND r2 == 2 possible?

No, if “sequentially consistent”

SC Example

Initially, x == y == 0

`2: y = 2;`	`4: x = 1;`
`1: r1 = x;`	`3: r2 = y;`

Is r1 == 1 AND r2 == 2 possible?

Yes, if the compiler reorders 1/2 and/or 3/4

Sequentially Consistent Java

Sequentially Consistent Java

- Simple model:
  - Guard every memory access with `barriers` (fields and array elements)

Sequentially Consistent Java

- Simple model:
  - Guard every memory access with `barriers` (fields and array elements)
- Drawbacks:
  - `Barriers` introduce overhead
  - Prevents many standard optimizations

Sequentially Consistent Java

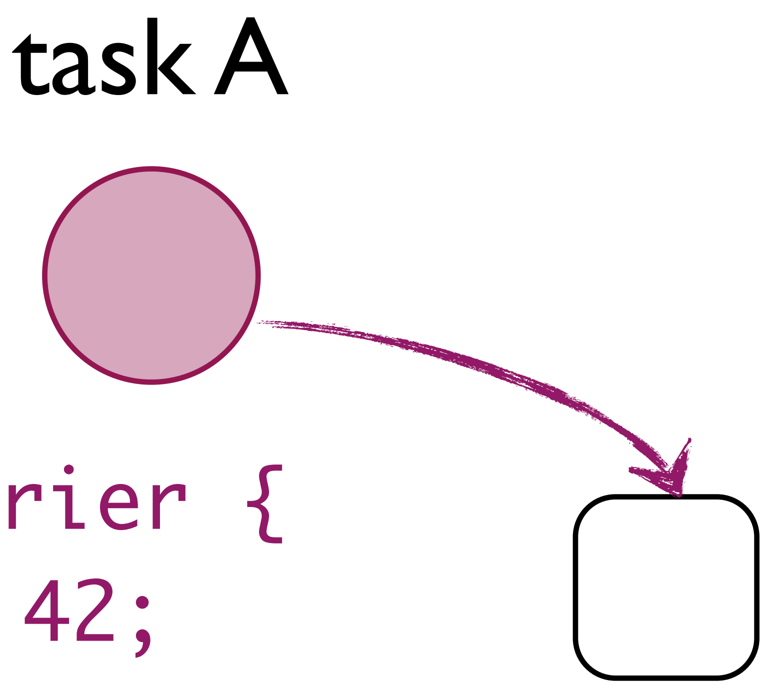
- Simple model:
  - Guard every memory access with `barriers` (fields and array elements)
- Drawbacks:
  - `Barriers` introduce overhead
  - Prevents many standard optimizations
- Optimize to re-gain performance:
  - Remove `barriers` where no parallel task may interfere

Optimization Question

- Can memory access `m1` interfere with `m2`?

Optimization Question

- Can memory access `m1` interfere with `m2`?



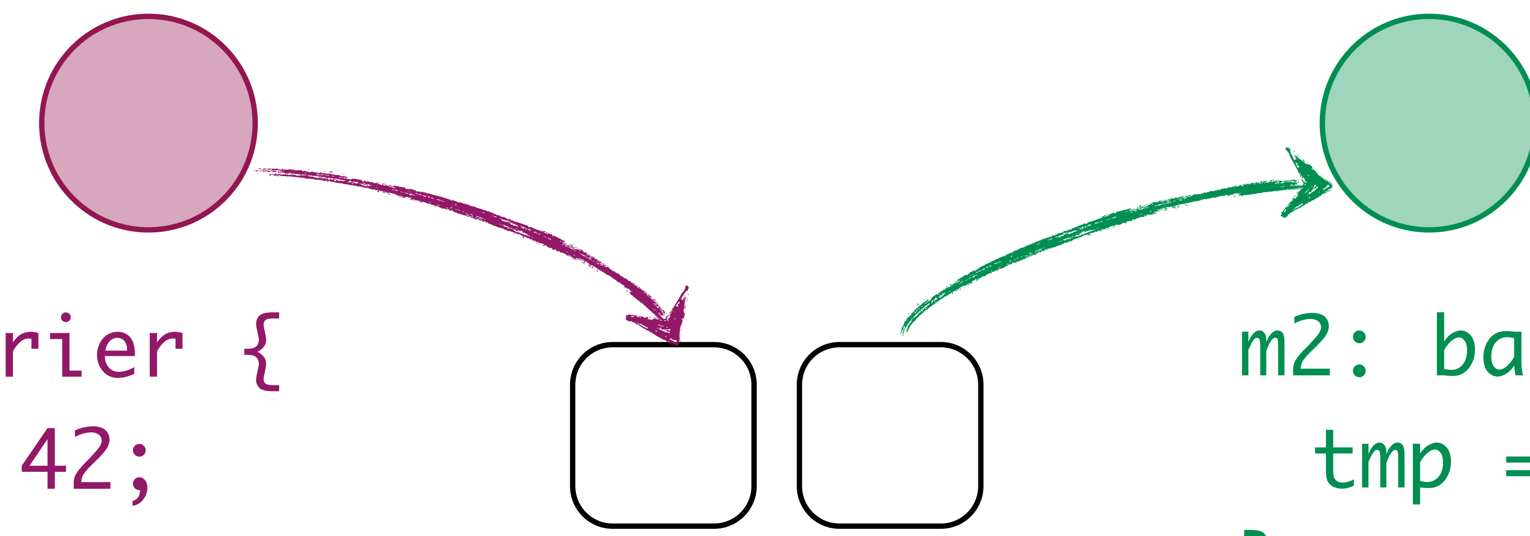

```
m1: barrier {
  x.f = 42;
}
```

Optimization Question

- Can memory access `m1` interfere with `m2`?

task A

task B

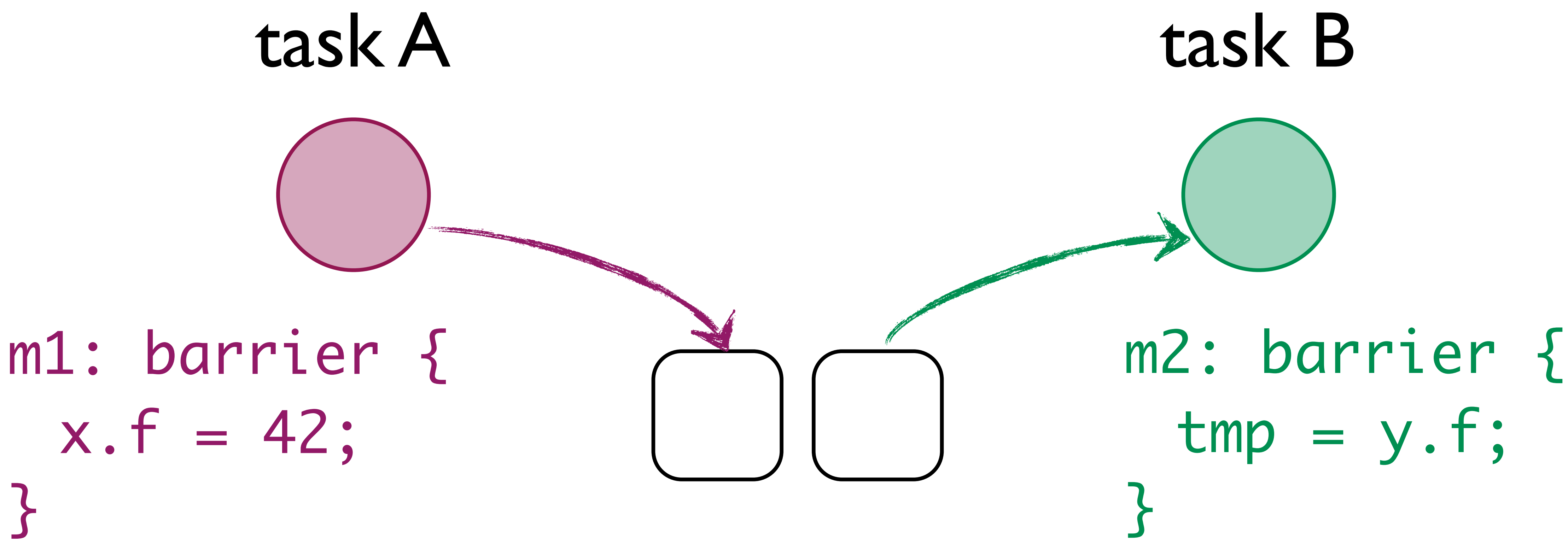


```
m1: barrier {
  x.f = 42;
}
```

```
m2: barrier {
  tmp = y.f;
}
```

Optimization Question

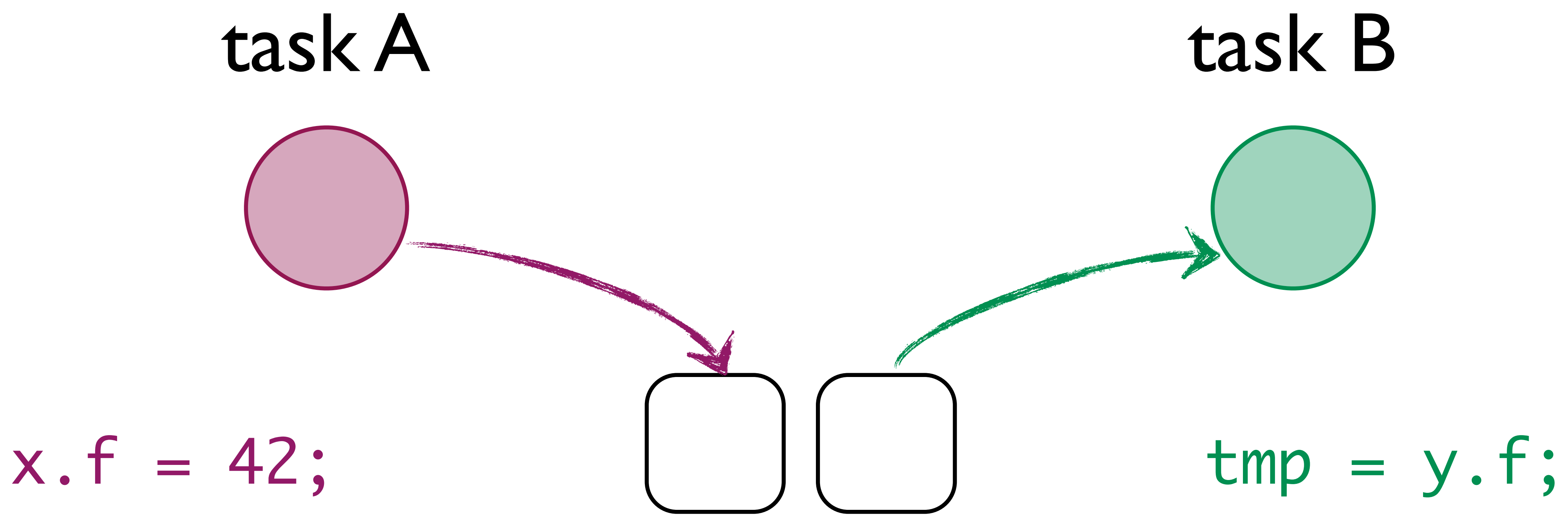
- Can memory access `m1` interfere with `m2`?



☑ Different Objects

Optimization Question

- Can memory access `m1` interfere with `m2`?



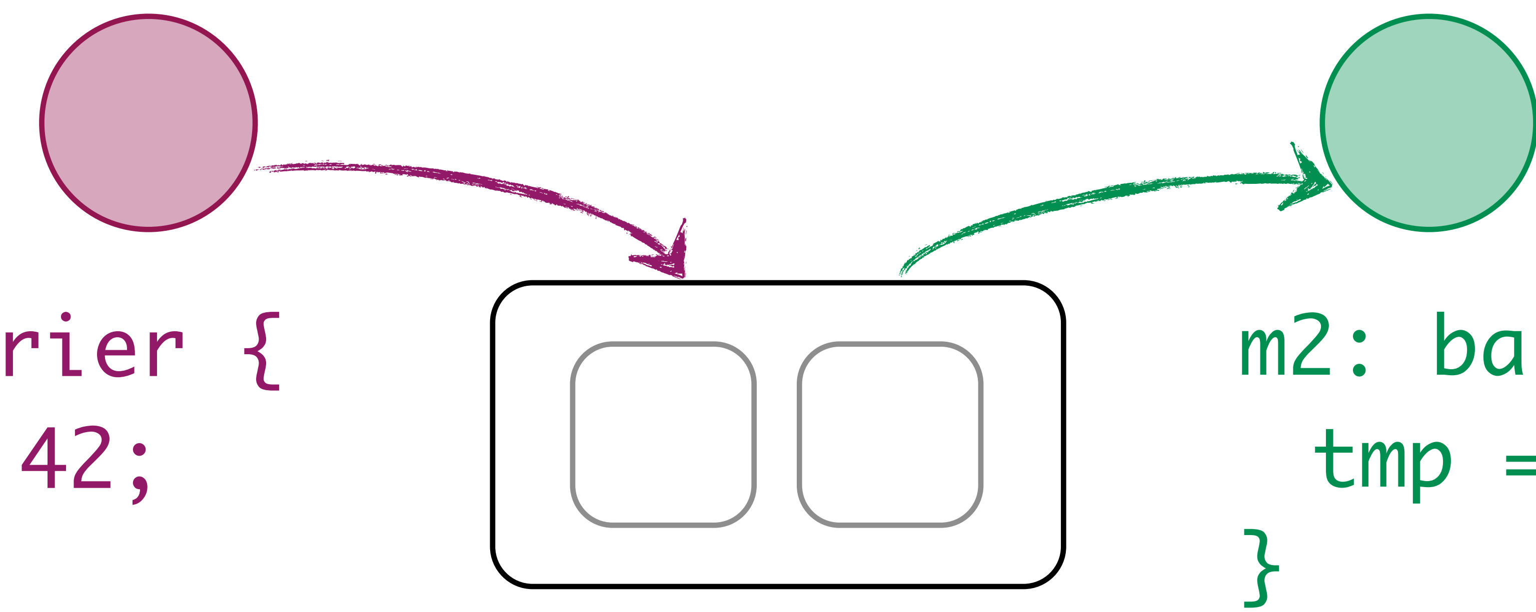
☑ Different Objects

Optimization Question

- Can memory access `m1` interfere with `m2`?

task A

task B

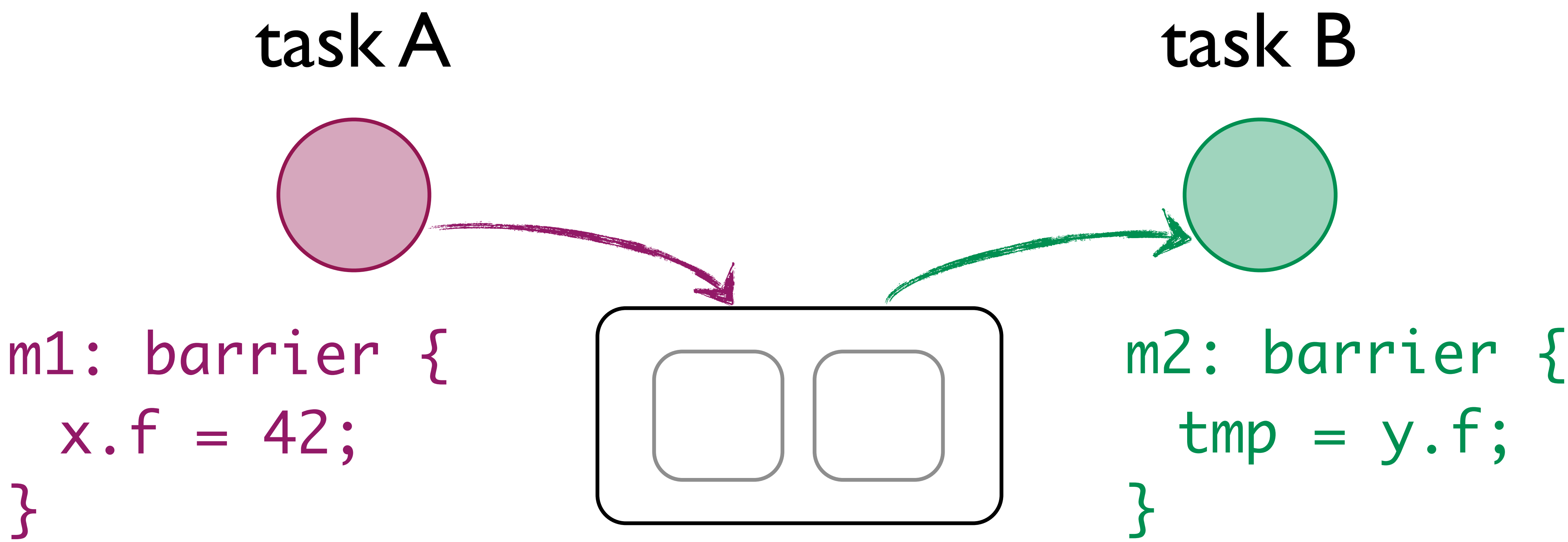


```
m1: barrier {
  x.f = 42;
}
```

```
m2: barrier {
  tmp = y.f;
}
```

Optimization Question

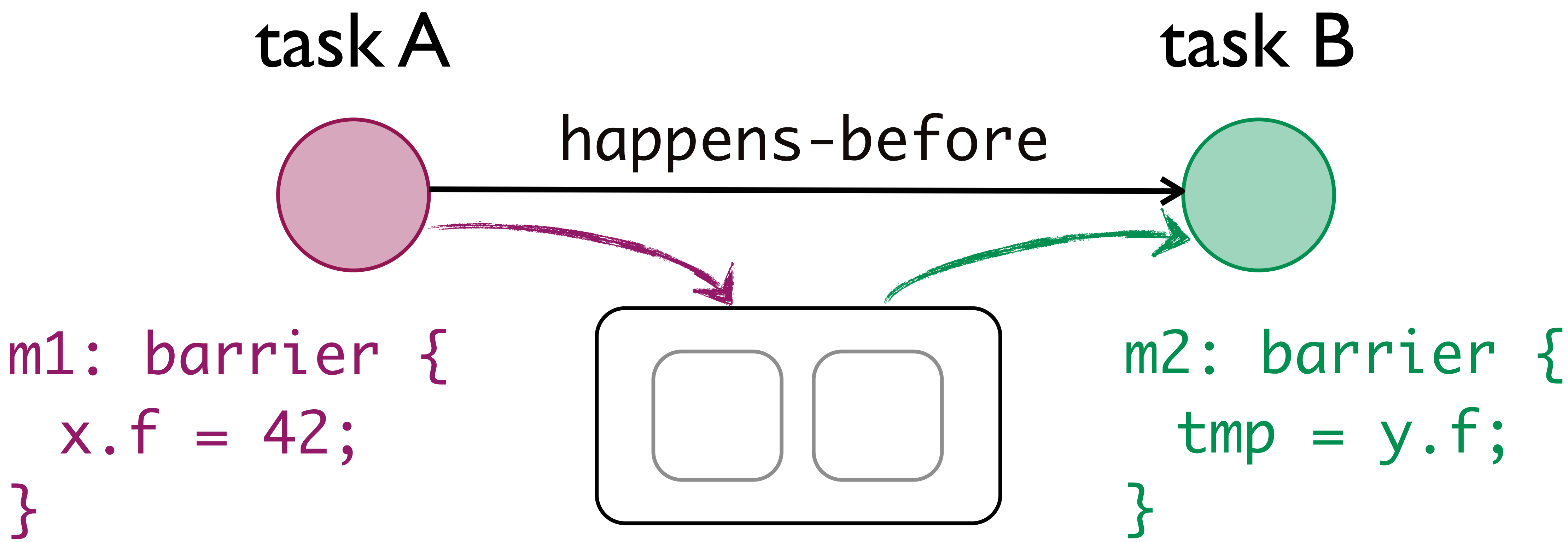
- Can memory access `m1` interfere with `m2`?



✗ Potentially aliased

Optimization Question

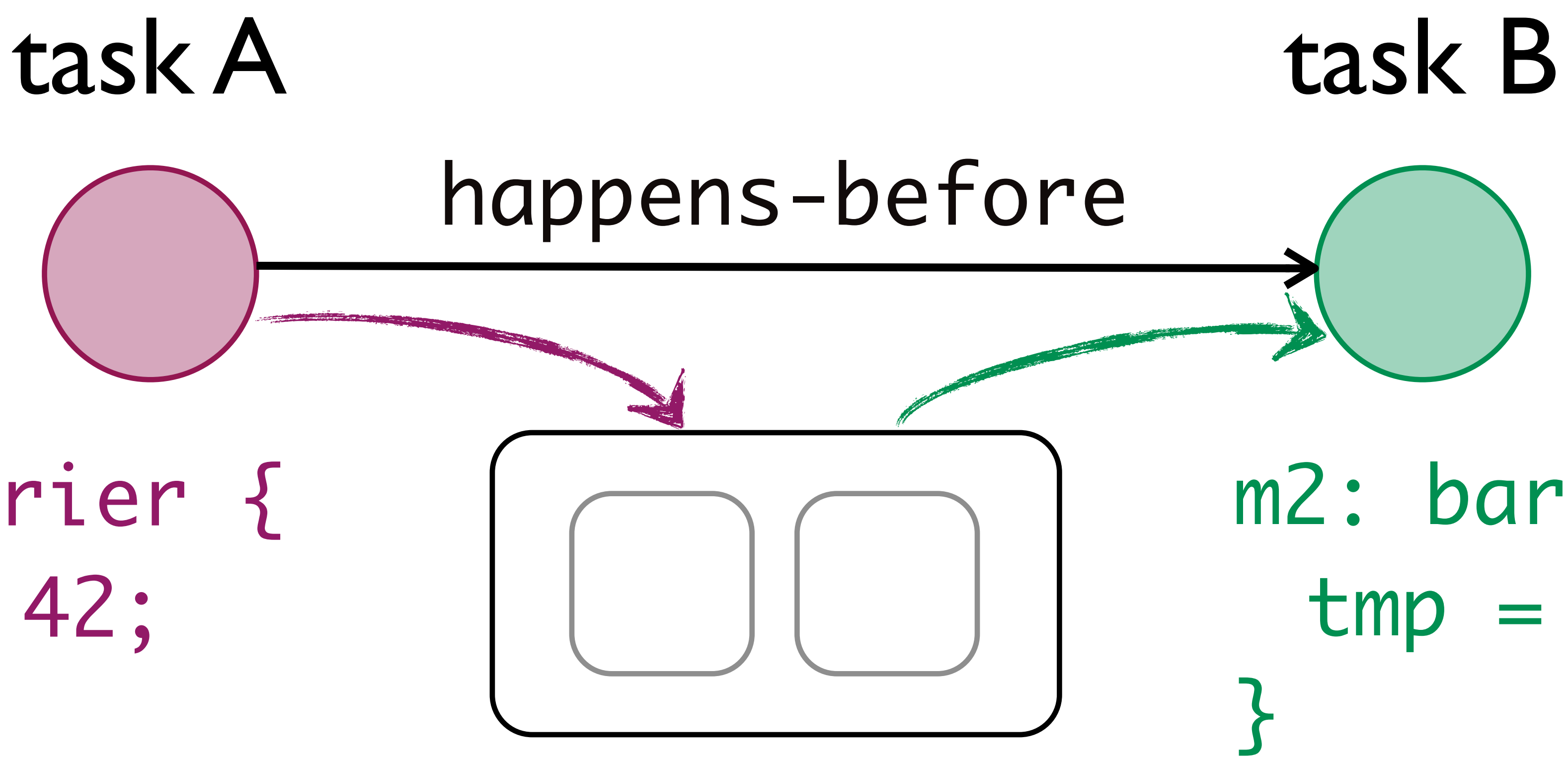
- Can memory access m1 interfere with m2?



✗ Potentially aliased

Optimization Question

- Can memory access `m1` interfere with `m2`?




```
m1: barrier {
  x.f = 42;
}
```

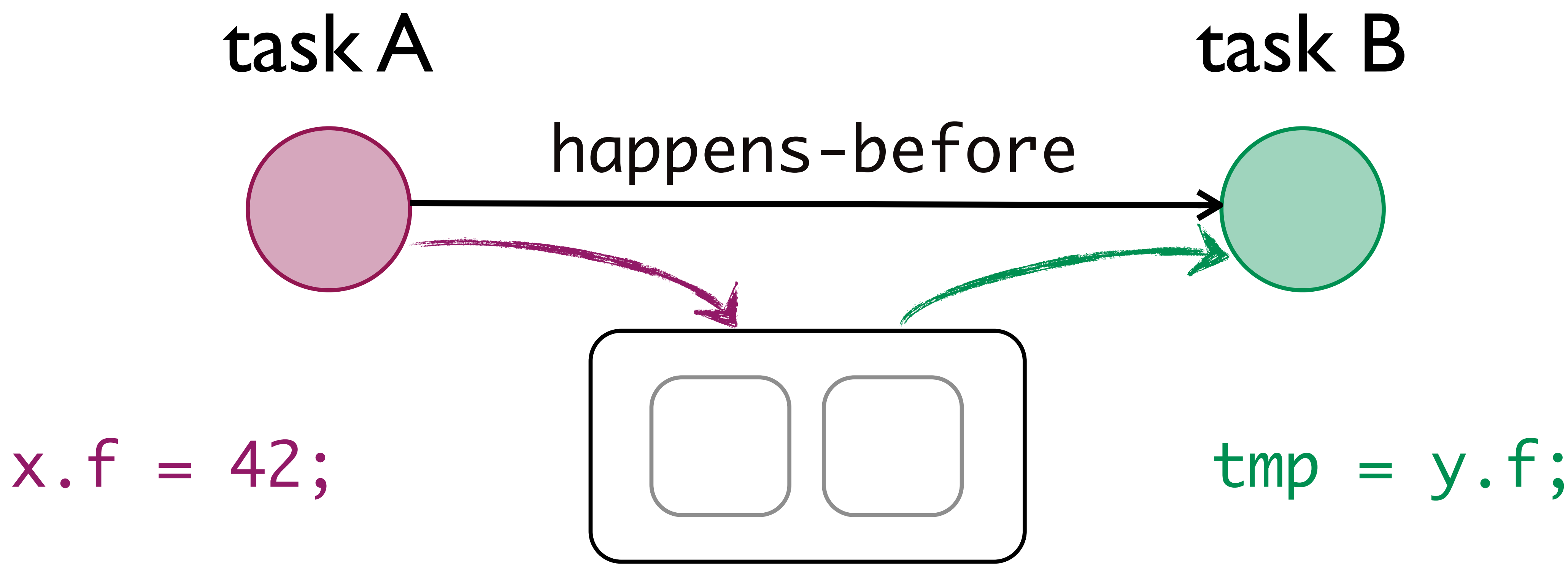
```
m2: barrier {
  tmp = y.f;
}
```

☒ Potentially aliased

☑ Ordered memory access

Optimization Question

- Can memory access m1 interfere with m2?



☒ Potentially aliased
☑ Ordered memory access

Sources of Task-Order Information

- Threads (Java)
  - Low-level synchronization, difficult to analyze

Sources of Task-Order Information

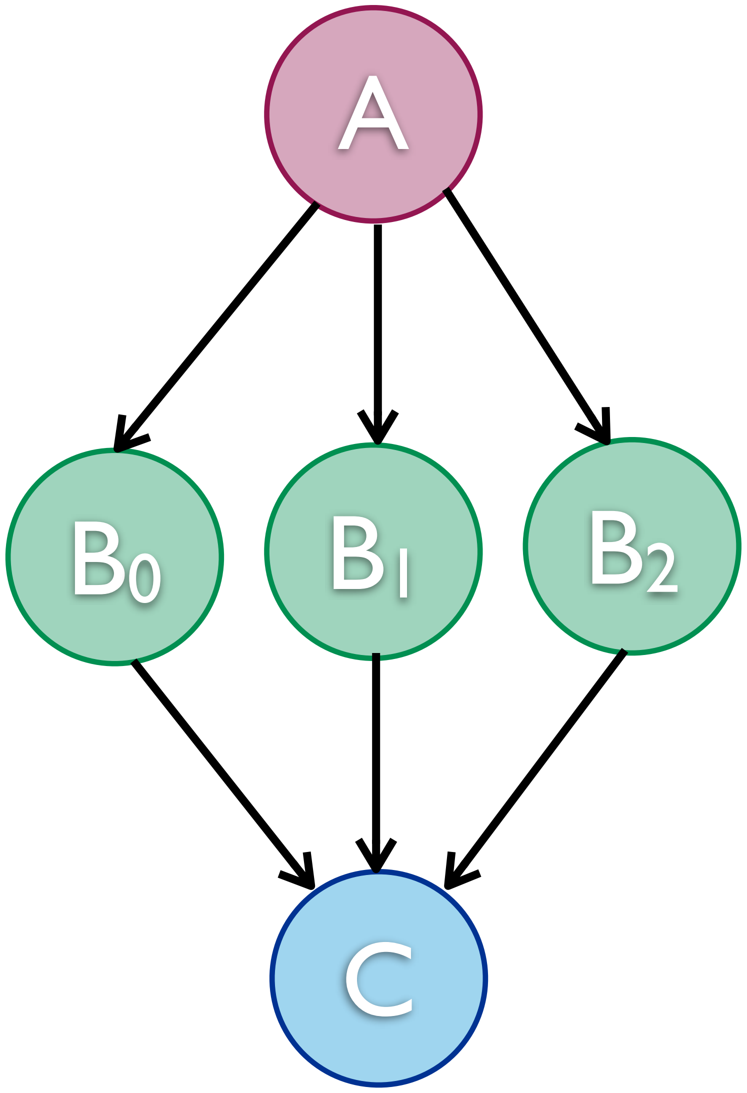
- Threads (Java)
  - Low-level synchronization, difficult to analyze
- Fork/join (OpenMP, X10, Cilk)
  - Lexical scoping simplifies analysis

Ordering in OpenMP

```
/*A*/

//#omp parallel for
for(int i=0; i<3; i++) {
  /*B*/
}

/*C*/
```

Ordering in OpenMP

```
/*A*/

//#omp parallel for
for(int i=0; i<3; i++) {
  /*B*/
}

/*C*/
```

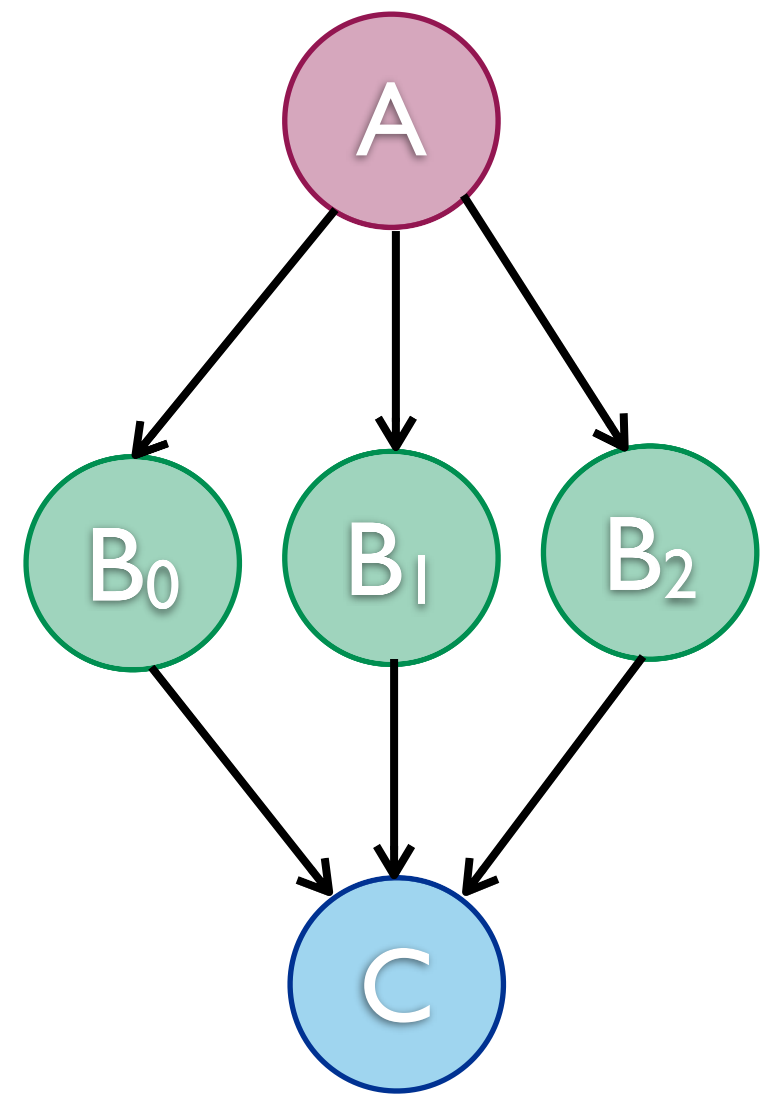


Sources of Task-Order Information

- Threads (Java)
  - Low-level synchronization, difficult to analyze
- Fork/join (OpenMP, X10, Cilk)
  - Lexical scoping simplifies analysis
- Task Libraries (Apple GCD, Microsoft TPL)
  - Feature explicit task ordering
  - Not much previous work here

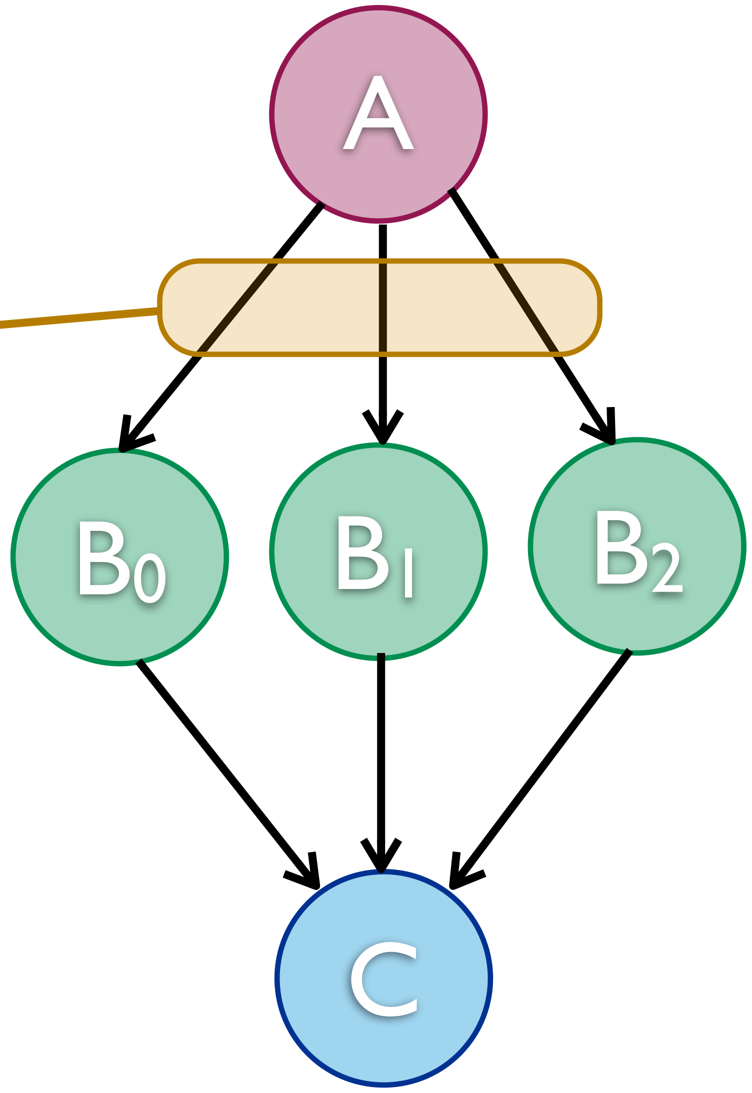
Ordering in Microsoft TPL

```
Task tA = Task.StartNew(/*A*/);

for(int i=0; i<3; i++) {
  tA.ContinueWith(/*B*/,
                  AttachedToParent);
}

/*C*/
```



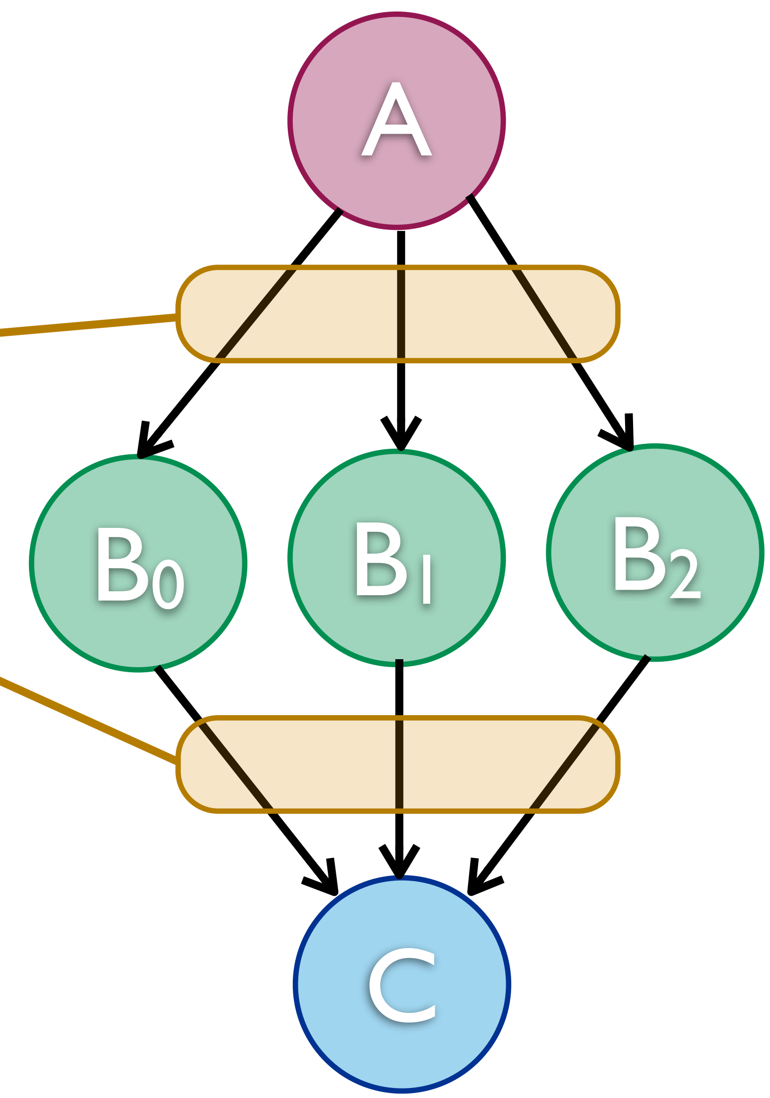
Ordering in Microsoft TPL

```
Task tA = Task.StartNew(/*A*/);

for(int i=0; i<3; i++) {
  tA.ContinueWith(/*B*/,
                  AttachedToParent);
}

/*C*/
```



Ordering in Microsoft TPL

```
Task tA = Task.StartNew(/*A*/);

for(int i=0; i<3; i++) {
  tA.ContinueWith(/*B*/,
                  AttachedToParent);
}

/*C*/
```



Sources of Task-Order Information

- Threads (Java)
  - Low-level synchronization, difficult to analyze
- Fork/join (OpenMP, X10, Cilk)
  - Lexical scoping simplifies analysis
- Task Libraries (Apple GCD, Microsoft TPL)
  - Feature explicit task ordering
  - Not much previous work here

Sources of Task-Order Information

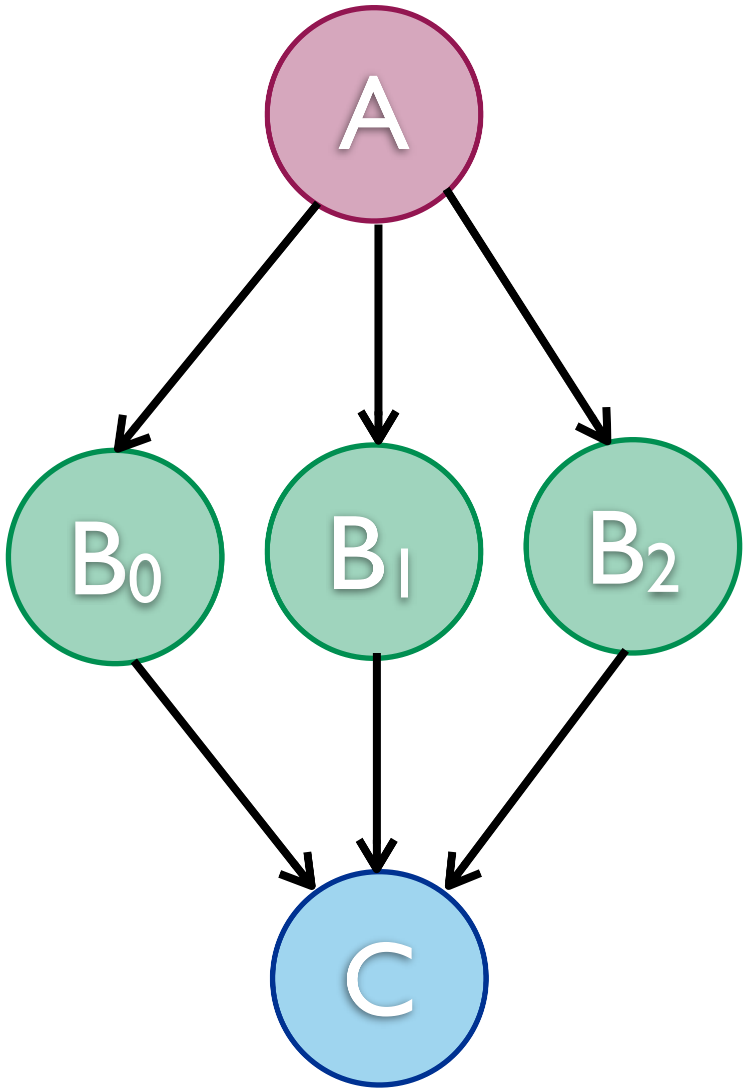
- Threads (Java)
  - Low-l … t to analyze
- Fork/joi …
  - Lexica …
- Task Lib … TPL)
  - Feature explicit task ordering
  - Not much previous work here

Abstraction:

Tasks with explicit happens-before relationships

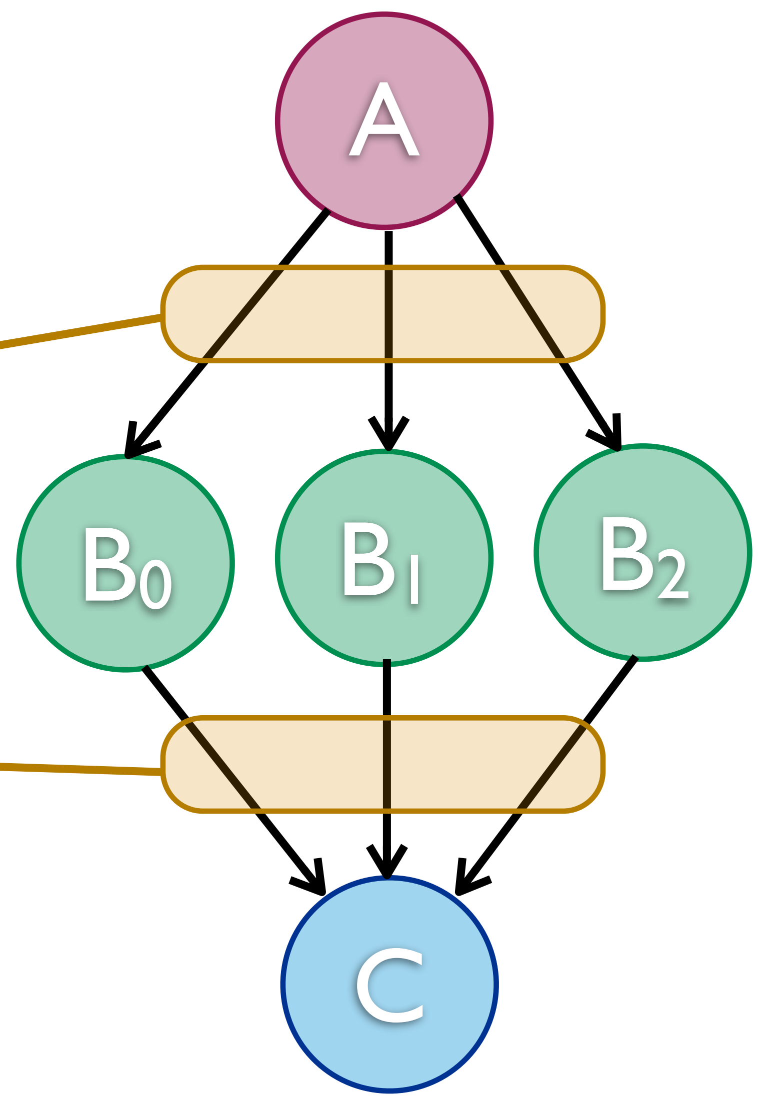
Our Model: Explicit Scheduling

```
Task a = schedule /*A*/;
Task c = schedule /*C*/;

for(int i=0; i<3; i++) {
  Task b = schedule /*B*/;
  a → b;
  b → c;
}
```



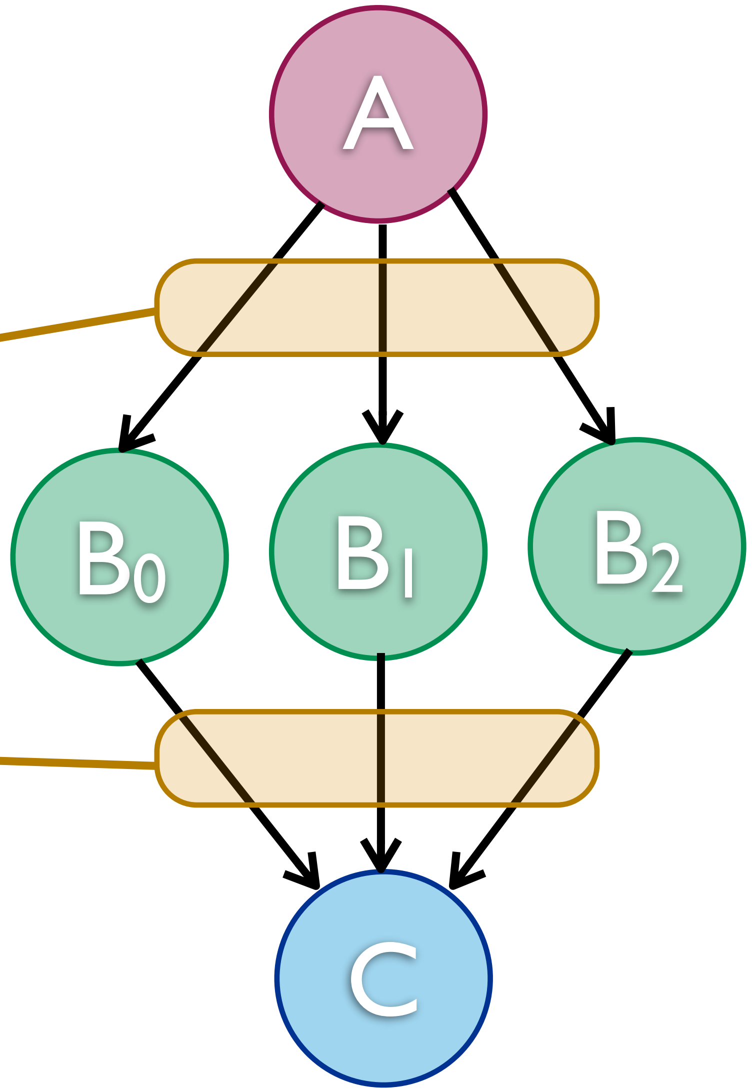
Our Model: Explicit Scheduling

```
Task a = schedule /*A*/;
Task c = schedule /*C*/;

for(int i=0; i<3; i++) {
  Task b = schedule /*B*/;
  a → b;
  b → c;
}
```



Our Model: Explicit Scheduling

```
Task a = schedule /*A*/;
Task c = schedule /*C*/;

for(int i=0; i<3; i++) {
  Task b = schedule /*B*/;
  a → b;
  b → c;
}
```



General enough to express threads, fork/join, thread libraries, ...

Schedule Analysis

- Static bytecode analysis
- Computes relation:
  `mayBeParallel(task1, task2)`
- If `!mayBeParallel(task1, task2)` then `task1` and `task2` are guaranteed to be ordered

Schedule Analysis Steps

Schedule Analysis Steps

1. Extract partial schedules from source code
   - ➡ Task variables plus ordering and loop information

Schedule Analysis Steps

1. Extract partial schedules from source code

   ➡Task variables plus ordering and loop information

2. `Callgraph` to resolve virtual entry points for tasks

Schedule Analysis Steps

1. Extract partial schedules from source code
   ➡Task variables plus ordering and loop information
2. `Callgraph` to resolve virtual entry points for tasks
3. Find tasks that may be created directly or indirectly `without ordering`

Schedule Analysis Steps

1. Extract partial schedules from source code
   ➡Task variables plus ordering and loop information
2. `Callgraph` to resolve virtual entry points for tasks
3. Find tasks that may be created directly or indirectly `without ordering`

- Key insight:
  - We look for `unordered-ness` not `ordered-ness`
  - Unordered-ness is monotonic!

On Monotonicity

```
task T1
  schedules A and B
  with A → B


task T2
  schedules A and B
```

On Monotonicity

```
task T1
  schedules A and B
  with A → B

task T2
  schedules A and B
```

Ordered-ness:

On Monotonicity

```
task T1
  schedules A and B
  with A → B


task T2
  schedules A and B
```

Ordered-ness:

@T1: A → B

On Monotonicity

```
task T1
  schedules A and B
  with A → B


task T2
  schedules A and B
```

Ordered-ness:

@T1: A → B

@T2: ∅

On Monotonicity

```
task T1
  schedules A and B
  with A → B


task T2
  schedules A and B
```

Ordered-ness:

@T1: ~~A → B~~

@T2: ∅

not monotonic

On Monotonicity

```
task T1
  schedules A and B
  with A → B

task T2
  schedules A and B
```

Ordered-ness:

@T1: ~~A → B~~

@T2: ∅

not monotonic

Unordered-ness:

On Monotonicity

```
task T1
  schedules A and B
  with A → B

task T2
  schedules A and B
```

not monotonic

Ordered-ness:

@T1: ~~A → B~~

@T2: ∅

Unordered-ness:

@T1: ∅

On Monotonicity

```
task T1
  schedules A and B
  with A → B

task T2
  schedules A and B
```

not monotonic

Ordered-ness:

@T1: ~~A → B~~

@T2: ∅

Unordered-ness:

@T1: ∅

@T2: A || B

On Monotonicity

```
task T1
  schedules A and B
  with A → B

task T2
  schedules A and B
```

Ordered-ness:

@T1: ~~A → B~~

@T2: ∅

not monotonic

Unordered-ness:

@T1: ∅

@T2: A || B

monotonic

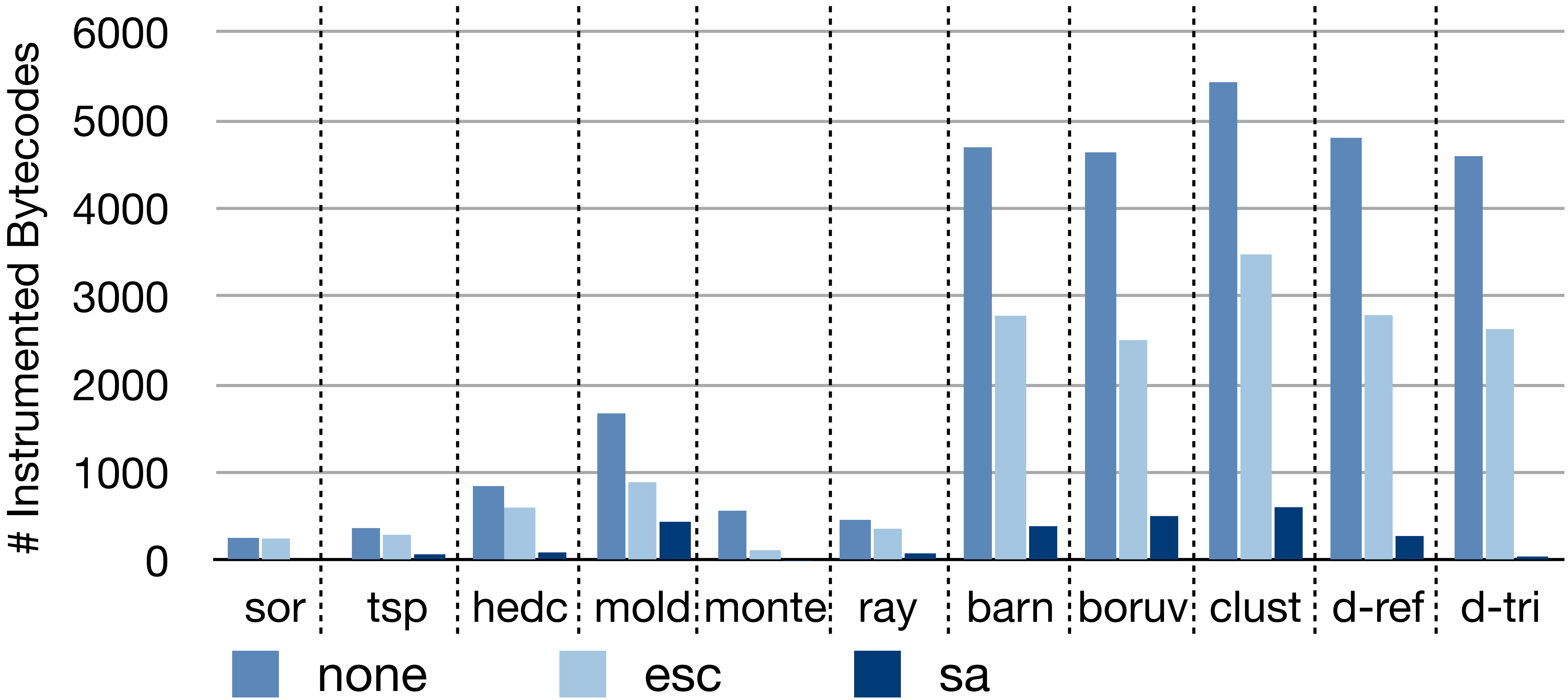
Evaluation Setup

- Bytecode-to-bytecode translation using `sun.misc.Unsafe.getXYVolatile()` and `sun.mist.Unsafe.putXYVolatile()` intrinsics
- Analyzes bytecode in SSA form
- Wala framework for analysis, Javassist for code generation
- Intel Core 2 Duo, 2.8 GHz, 4Gb RAM
  - ➡ 2 Java Threads
- Java 1.6.0 (Mac)

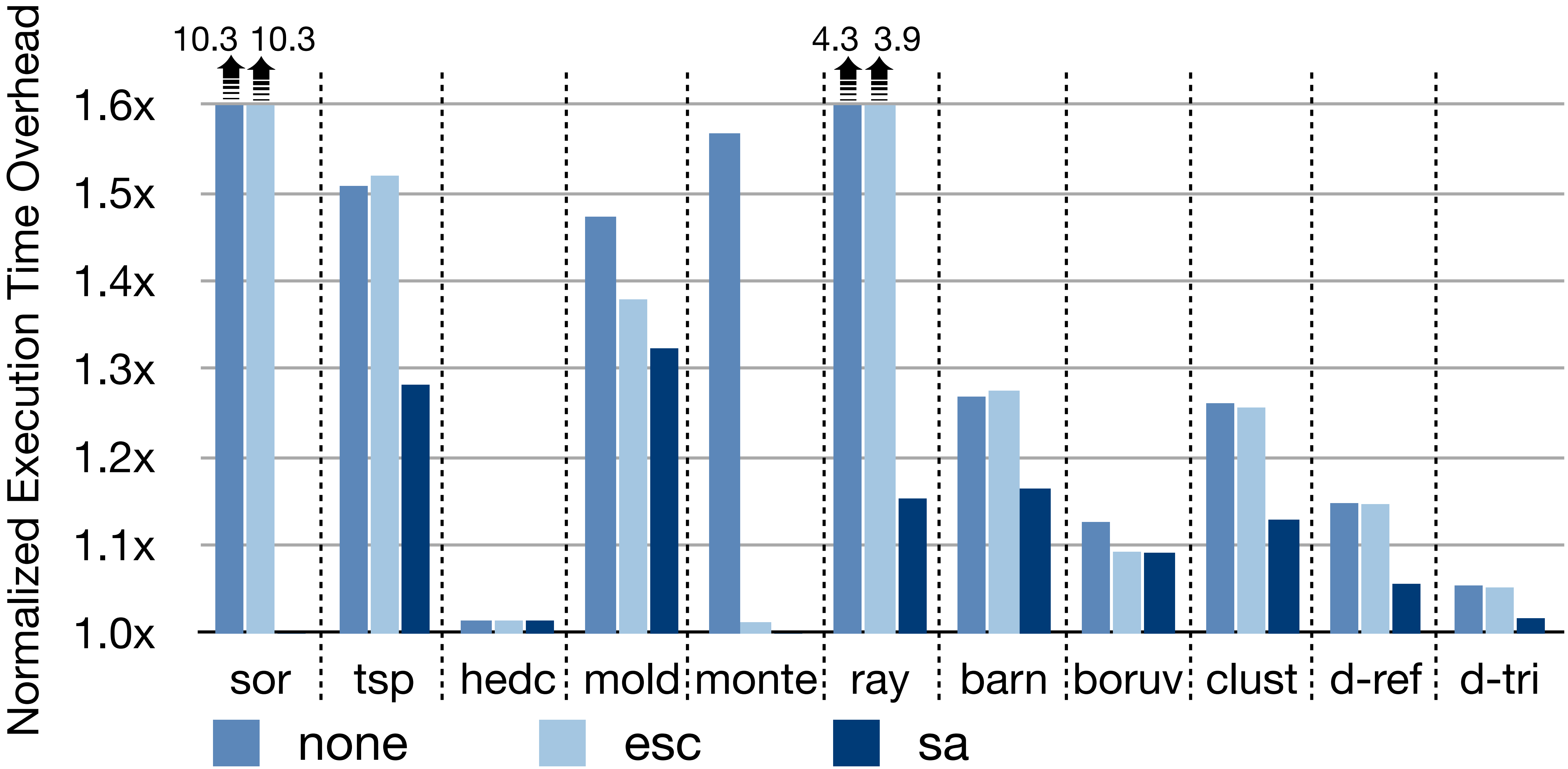
Benchmarks

- Erco Benchmarks: `sor`, `tsp`, `hedc`
- Java Grande (numeric apps): `mold`, `ray`, `monte`
- Lonestar (Galois): `barn`, `boruv`, `clust`, `d-tri`, `d-ref`
- Configurations:
  - Hand-optimized (`baseline`)
  - No optimizations (`none`)
  - Escape analysis only (`esc`)
  - Schedule analysis + escape analysis (`sa`)

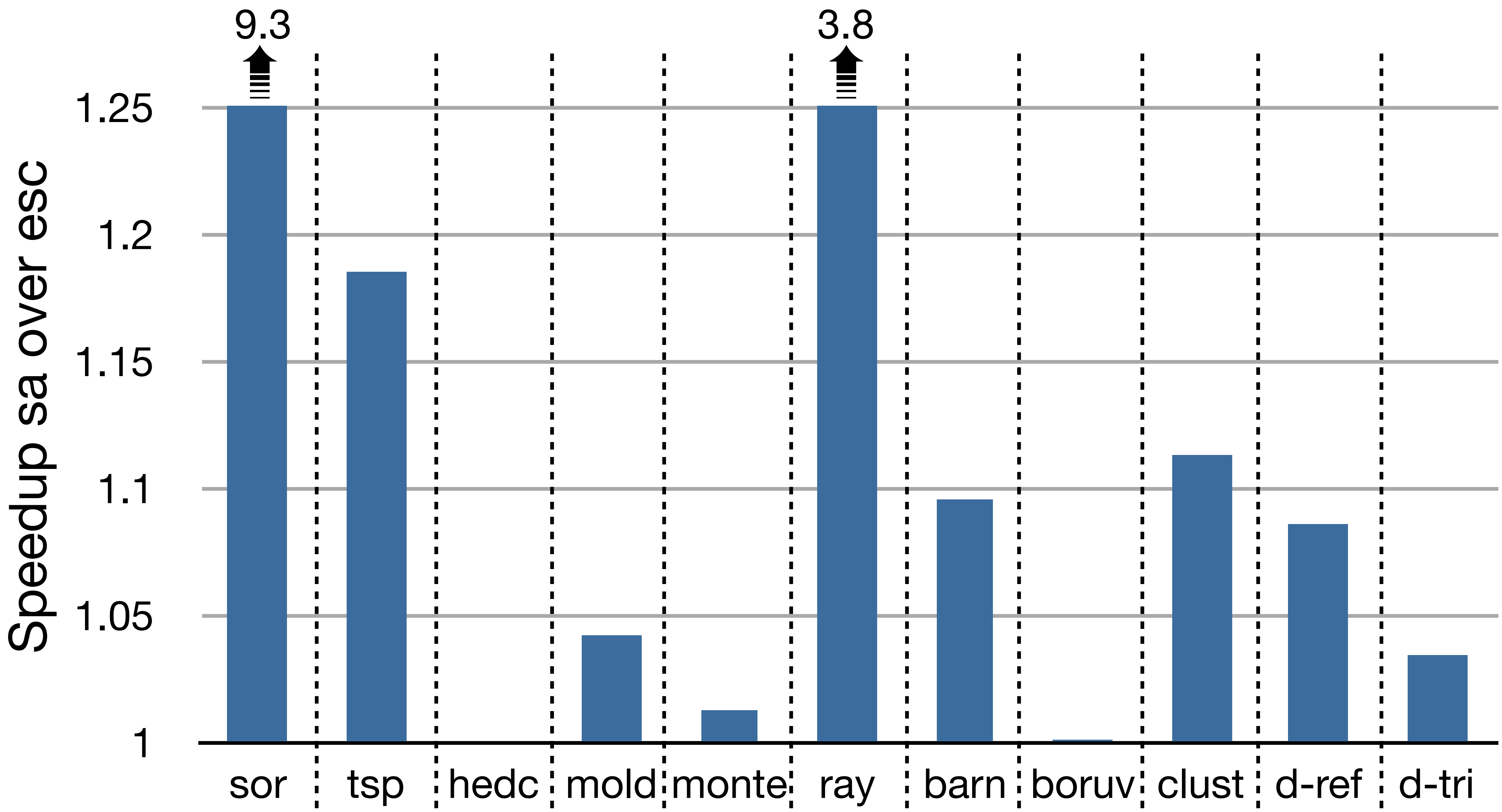
Instrumentation Overhead



Execution Time



Speedup sa vs. esc



Related Work

- **Compiler techniques for high performance sequentially consistent java programs.** Z. Sura et al, PPoPP'05
- **A case for an SC-preserving compiler.** D. Marino et al, PLDI'11
- **Efficient sequential consistency using conditional fences.** C. Lin, V Nagarajan, R. Gupta, PACT'10
- **BulkCompiler: high-performance sequential consistency through cooperative compiler and hardware support.** W. Ahn et al, MICRO'09
- **MHP Analysis.** (Agarwal et al)

Concluding Remarks

- Optimizations for `shared-memory` parallel programs need `task-order information` to be effective
- `Schedule analysis` is an approach that
  - can extract task-order information from `real-world` programs
  - provides starting point for optimizations
- Modest overhead over hand-optimized sequentially consistent programs