Exploiting Task-Order Information in Optimizing Compilers for Parallel Programs

Christoph Angerer

Motivation

- Parallel programming becomes increasingly mainstream
- Influences models, languages, runtime libraries
- Added support for:
  - Controlling access to shared data
  - Controlling parallel program flow

Motivation

- Parallel programming becomes increasingly mainstream
- Influences models, languages, runtime libraries
- Added support for:
  - Controlling access to shared data
  - Controlling parallel program flow

Motivation

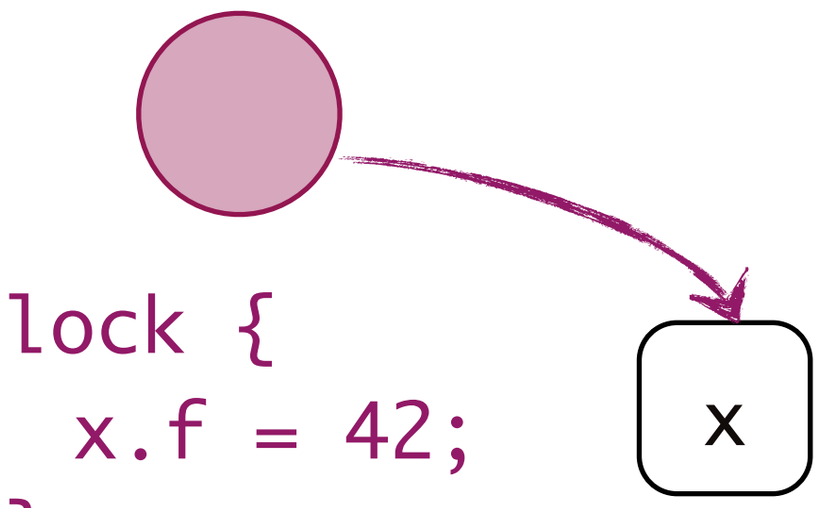
- Parallel programming becomes increasingly mainstream
- Influences models, languages, runtime libraries
- Added support for:
  - Controlling access to shared data
  - Controlling parallel program flow

Can a compiler exploit information about the execution-order of tasks?

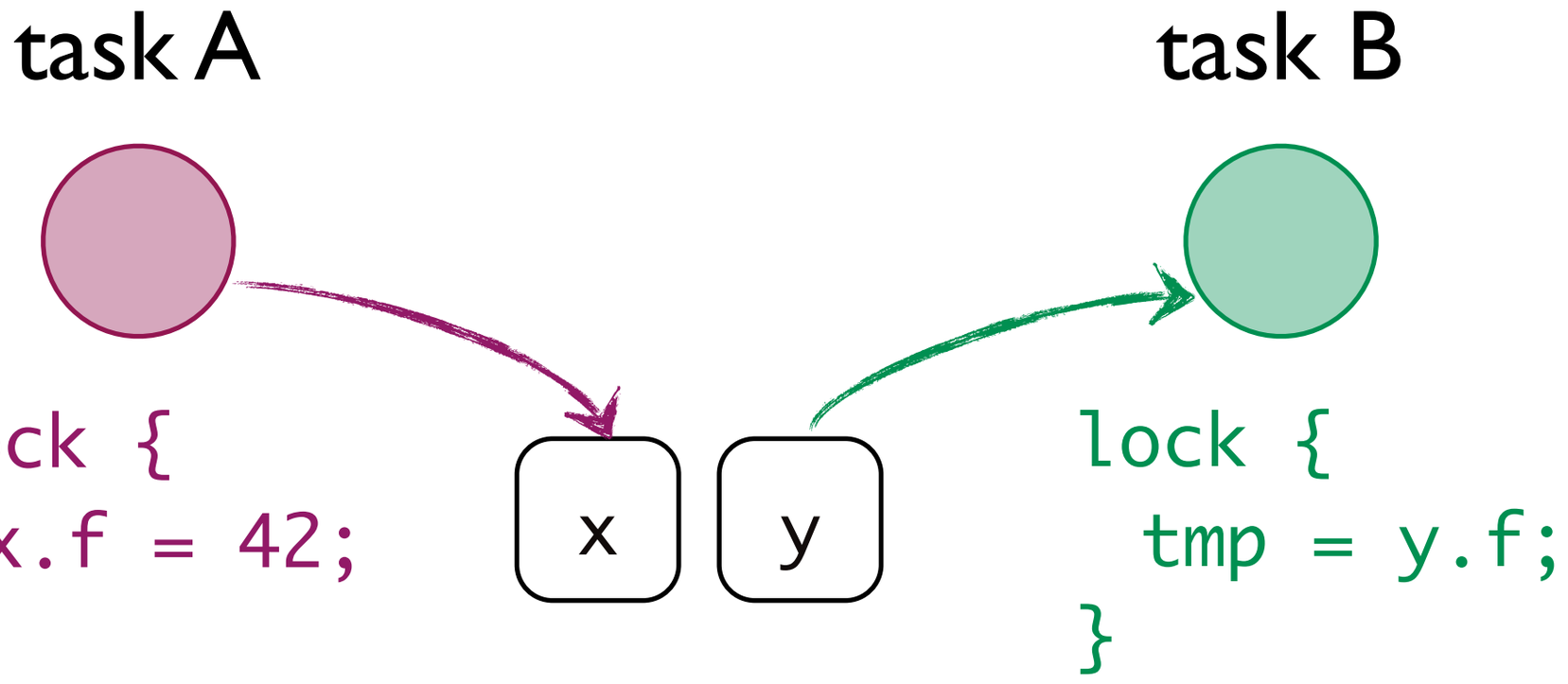
Example

Example

task A

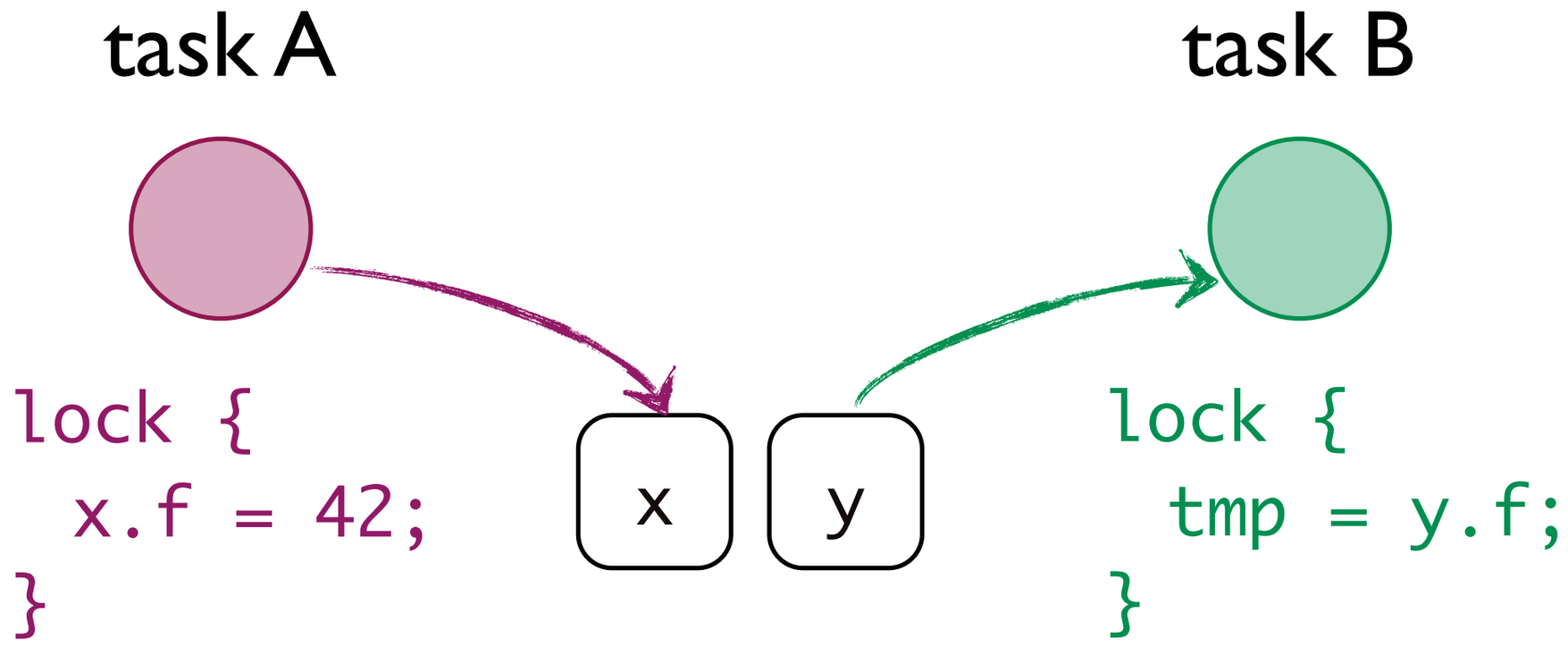


Example



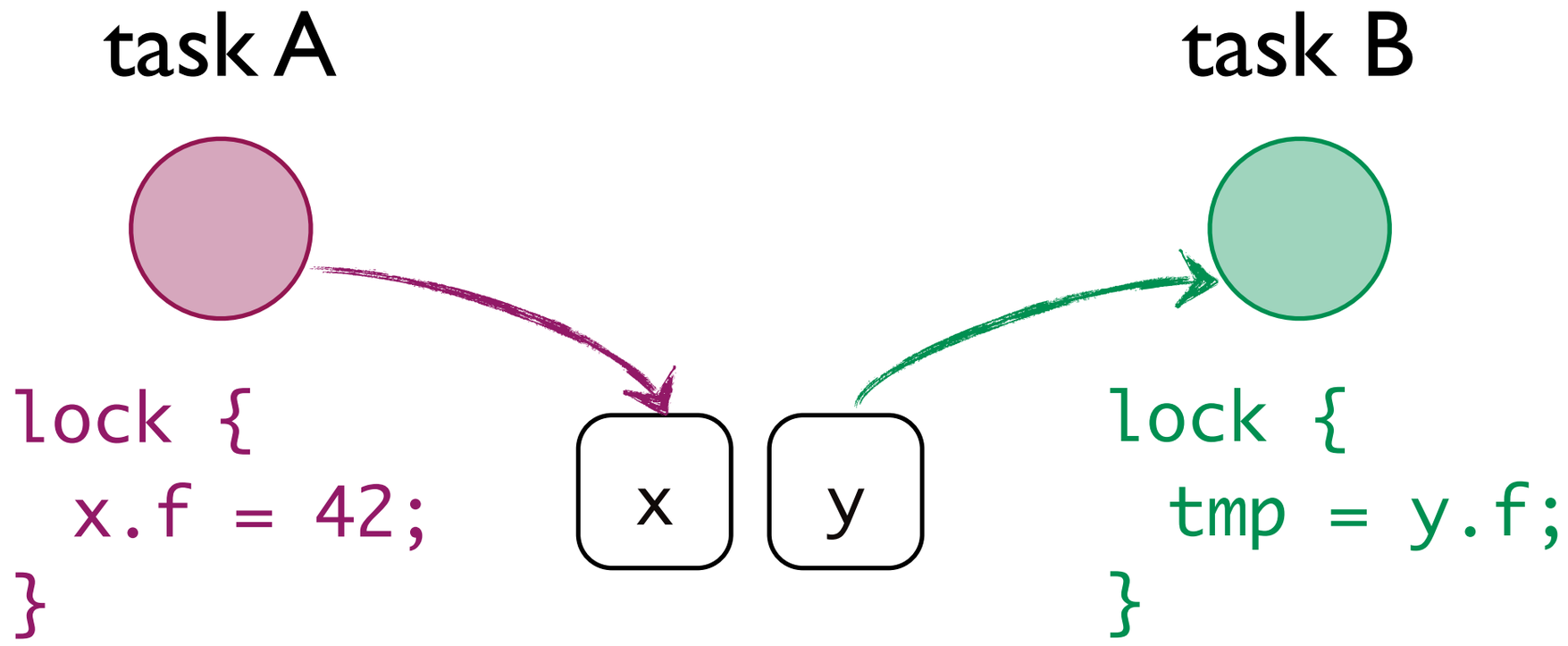
Example

- Can the compiler remove the lock?



Example

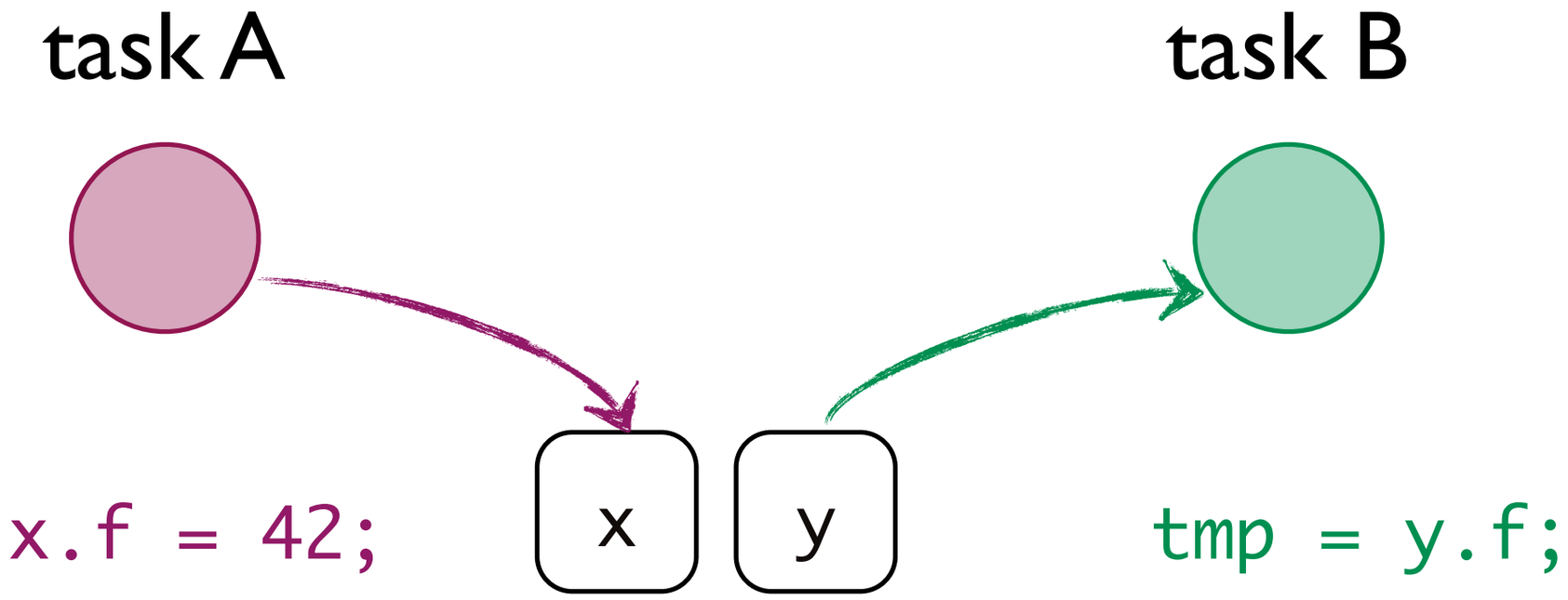
- Can the compiler remove the lock?



☑ Different Objects

Example

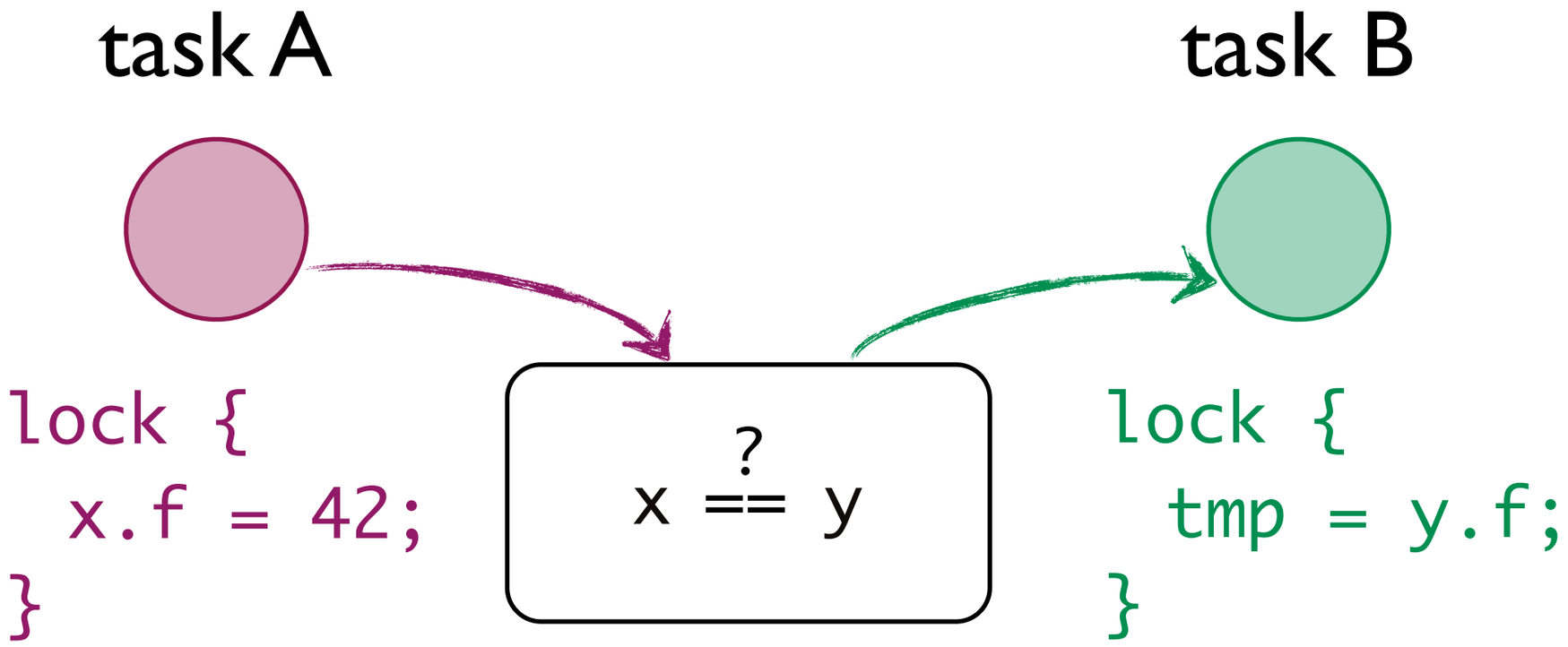
- Can the compiler remove the lock?



☑ Different Objects

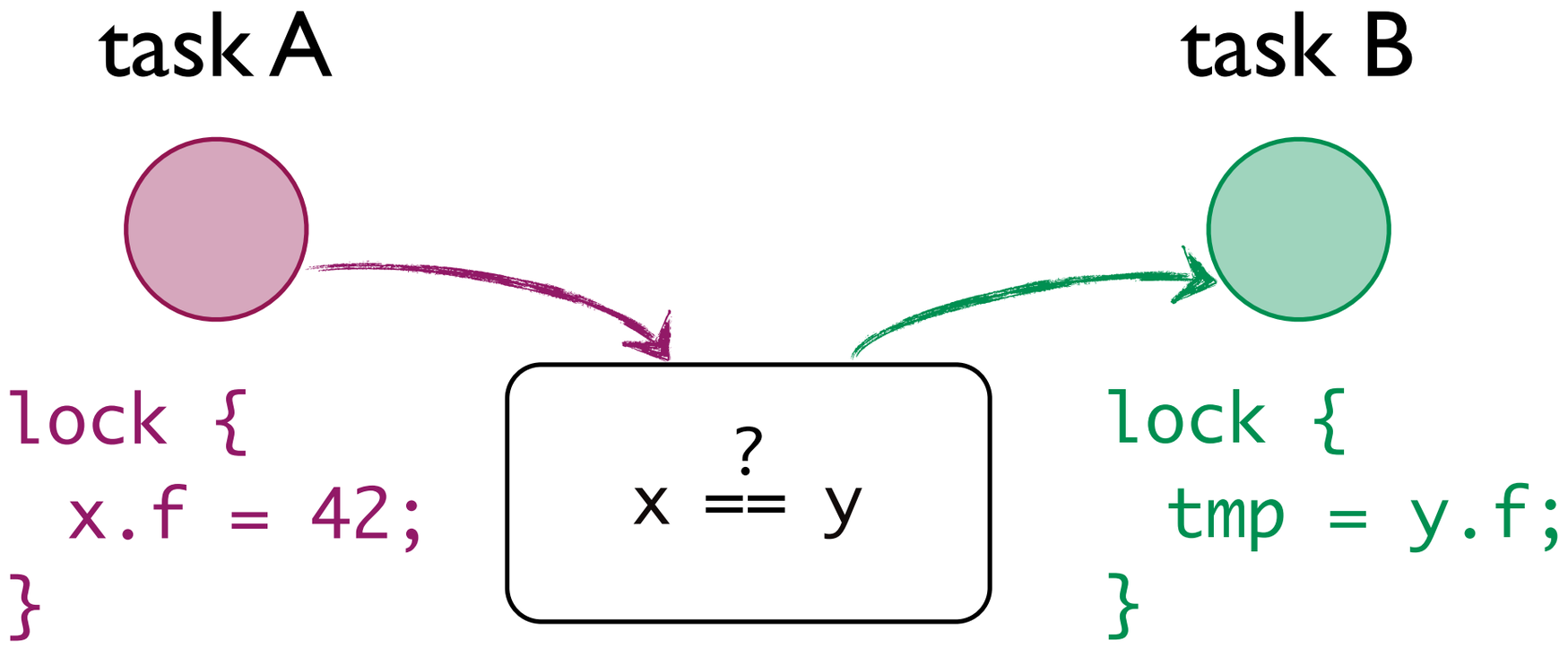
Example

- Can the compiler remove the lock?



Example

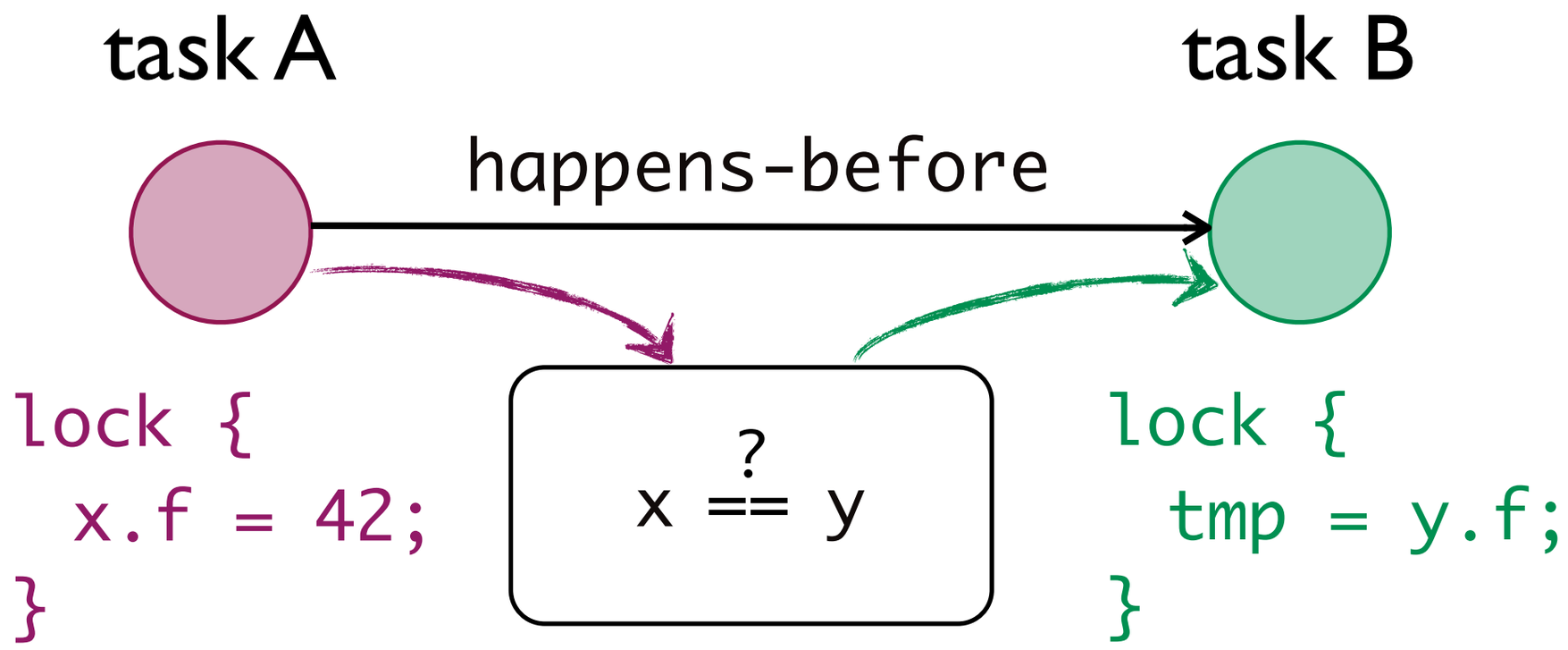
- Can the compiler remove the lock?



☒ Potentially aliased

Example

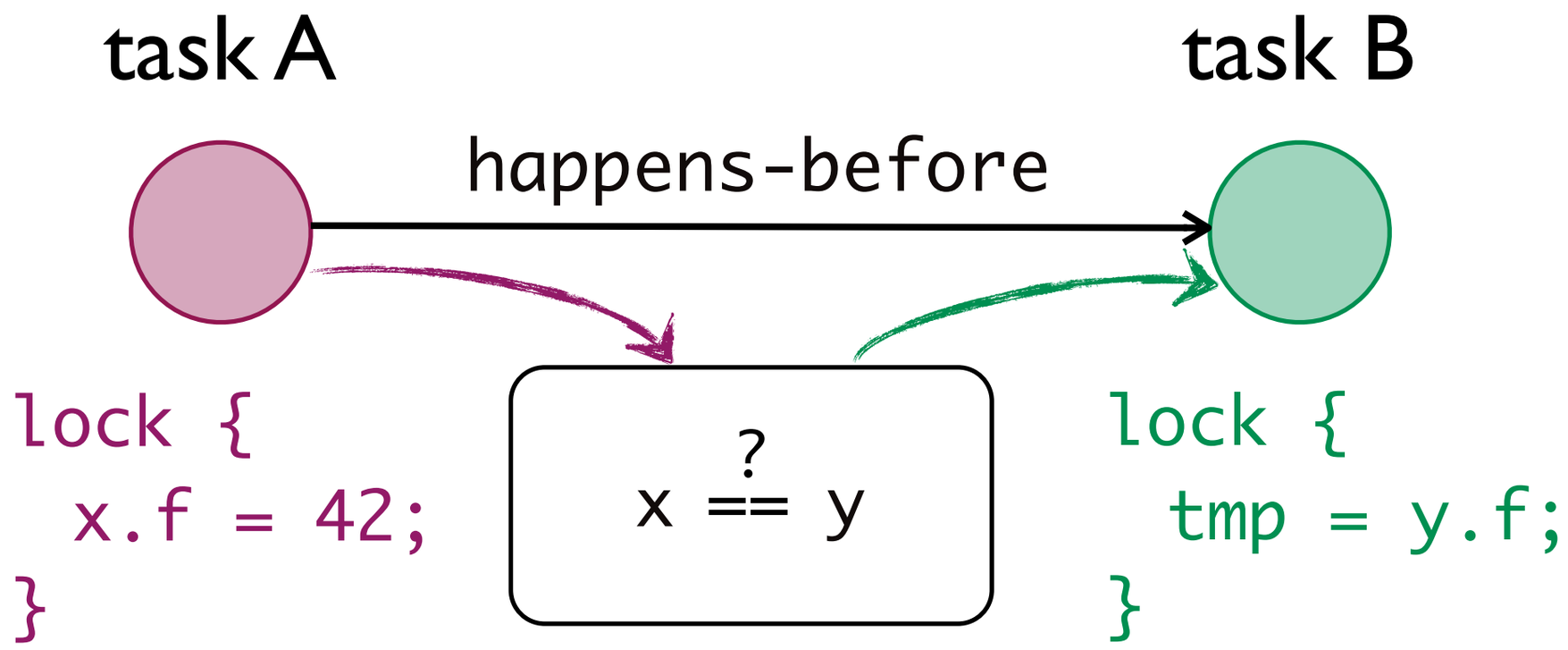
- Can the compiler remove the lock?



☒ Potentially aliased

Example

- Can the compiler remove the lock?

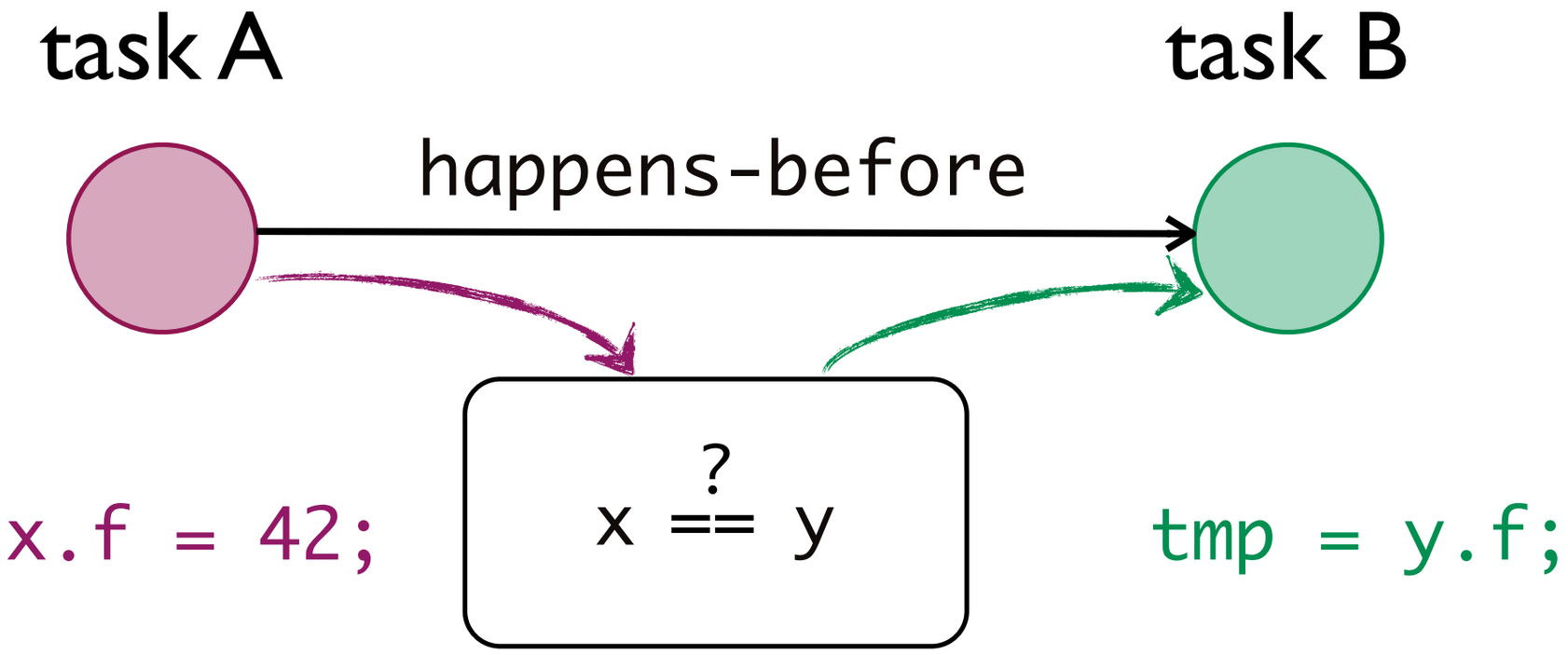


☒ Potentially aliased

☑ Ordered memory access

Example

- Can the compiler remove the lock?



☒ Potentially aliased

☑ Ordered memory access

Thesis Statement

"Effective optimizations for shared-memory parallel programs need task-order information."

Contributions

- Programming model with explicit task-ordering constraints
- Schedule analysis to extract task-order information
- Evaluation of compiler optimizations that use task-order information

Outline

- Motivation
- Task Model
- Schedule Analysis
- Two Case Studies
- Conclusion

Sources of Task-Ordering

Sources of Task-Ordering

- Threads (Java)
  - **Low-level signalling, difficult to analyze** [Barik 2006]

Sources of Task-Ordering

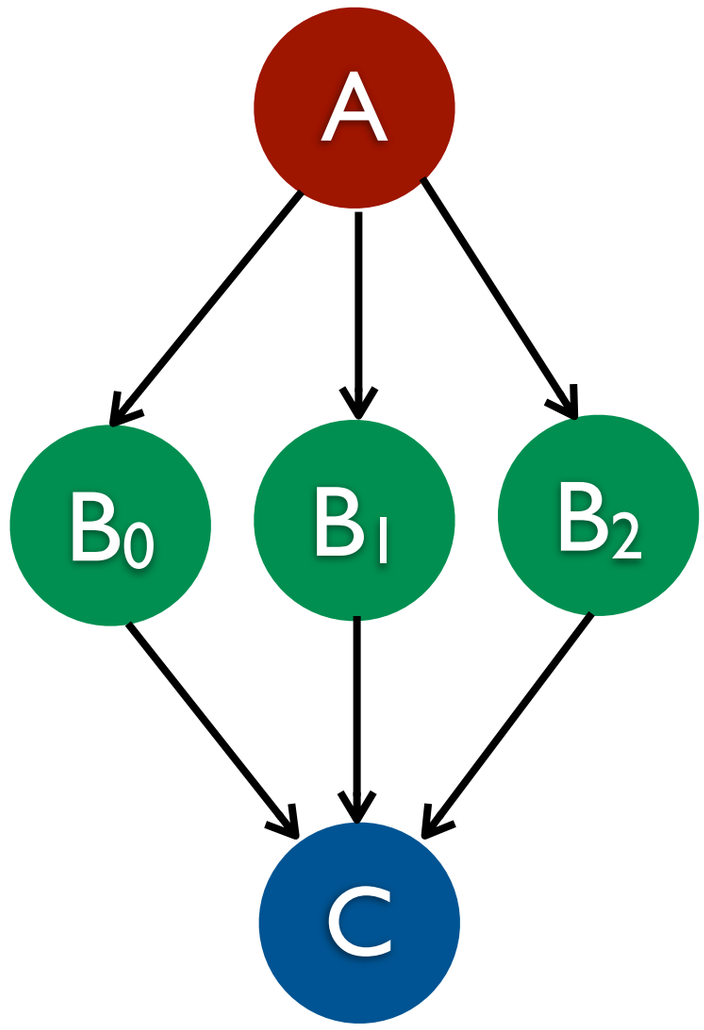
- Threads (Java)
  - **Low-level signalling, difficult to analyze** [Barik 2006]
- Fork/join (OpenMP, X10, Cilk)
  - Lexical scoping simplifies analysis

Sources of Task-Ordering

- Threads (Java)
  - Low-level signalling, difficult to analyze [Barik 2006]
- Fork/join (OpenMP, X10, Cilk)
  - Lexical scoping simplifies analysis
- Task Libraries (Apple GCD, Microsoft TPL, Intel TBB)
  - Feature explicit task ordering
  - Not much previous work here

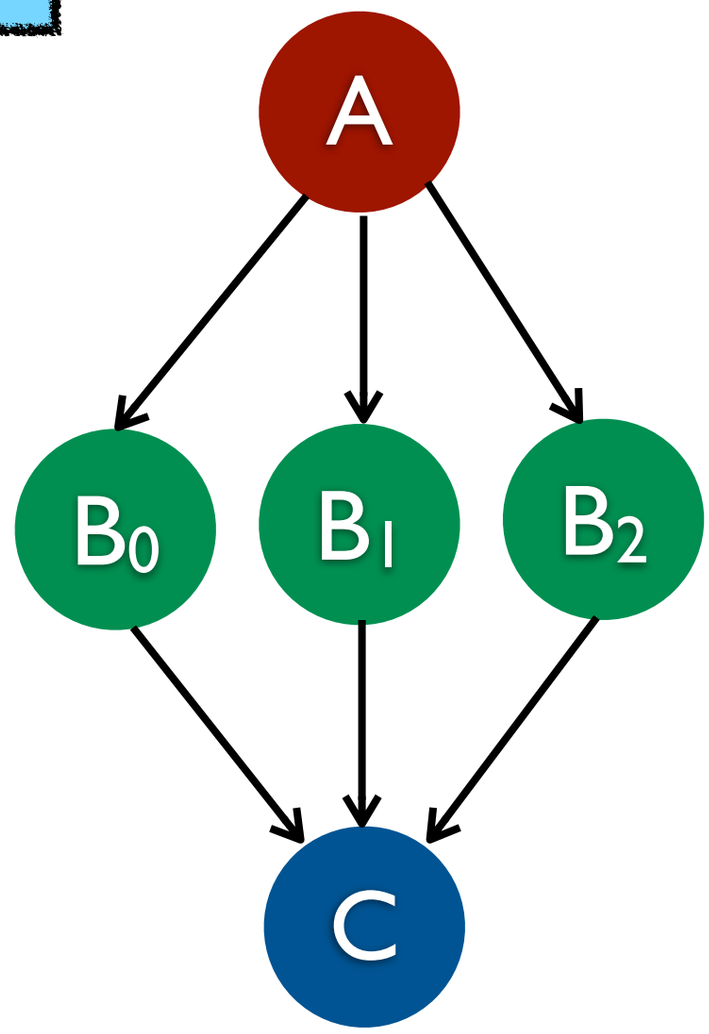
Ordering in Microsoft TPL

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

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

/*C*/
```



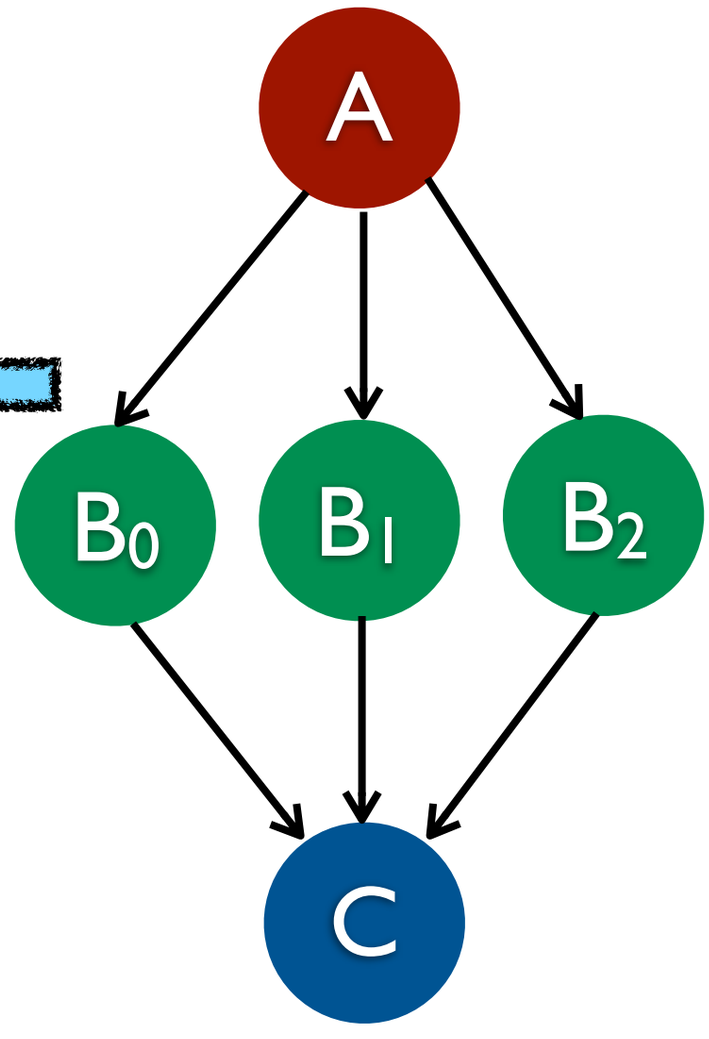
Ordering in Microsoft TPL

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

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

/*C*/
```



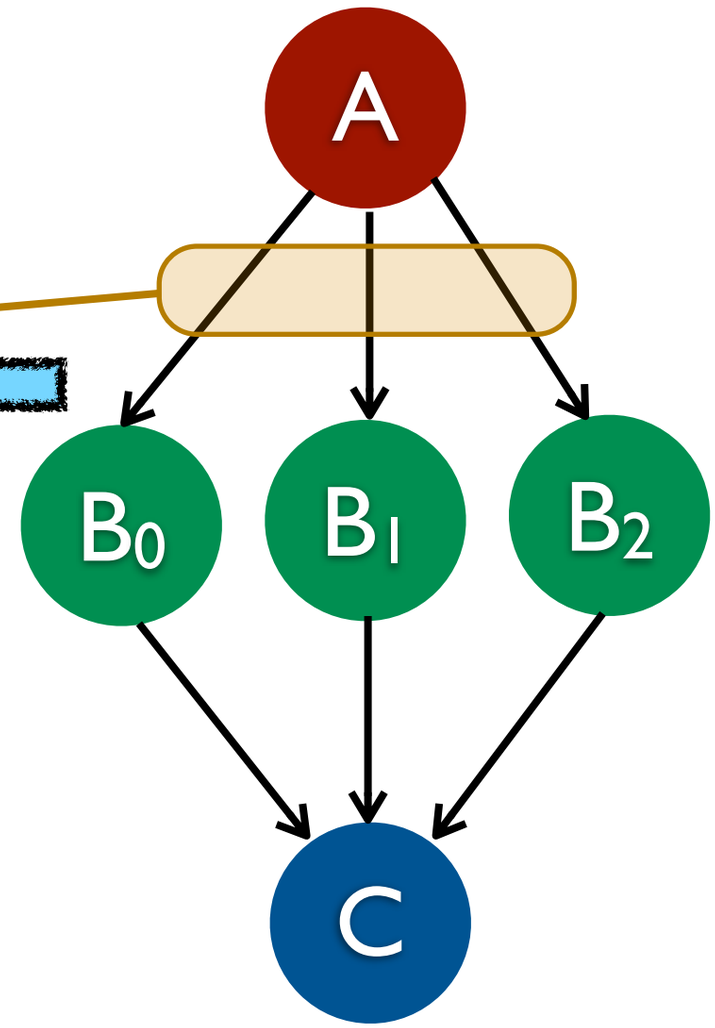
Ordering in Microsoft TPL

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

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

/*C*/
```



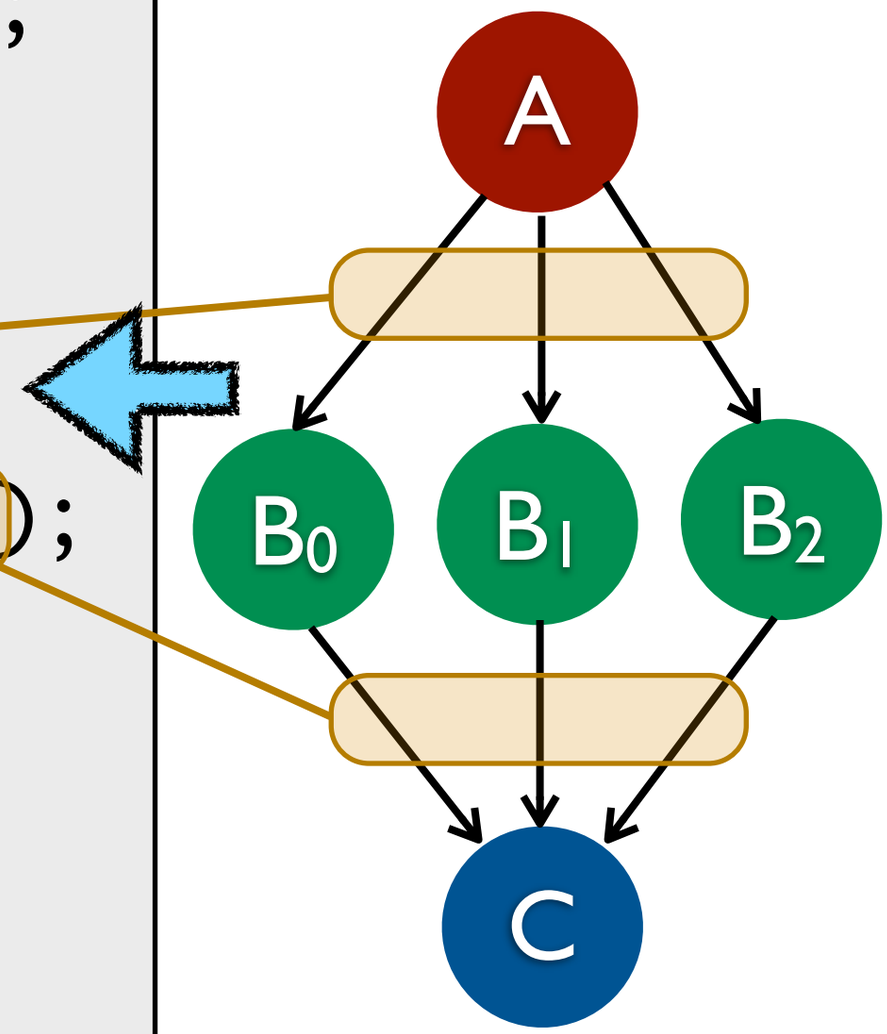
Ordering in Microsoft TPL

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

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

/*C*/
```



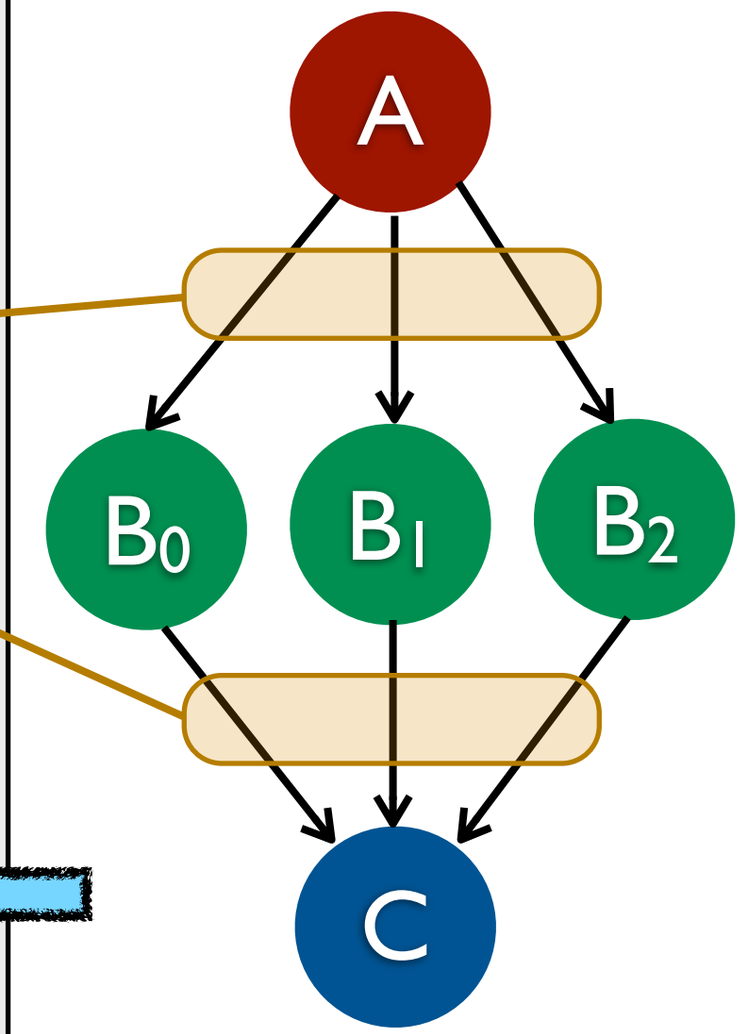
Ordering in Microsoft TPL

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

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

/*C*/
```



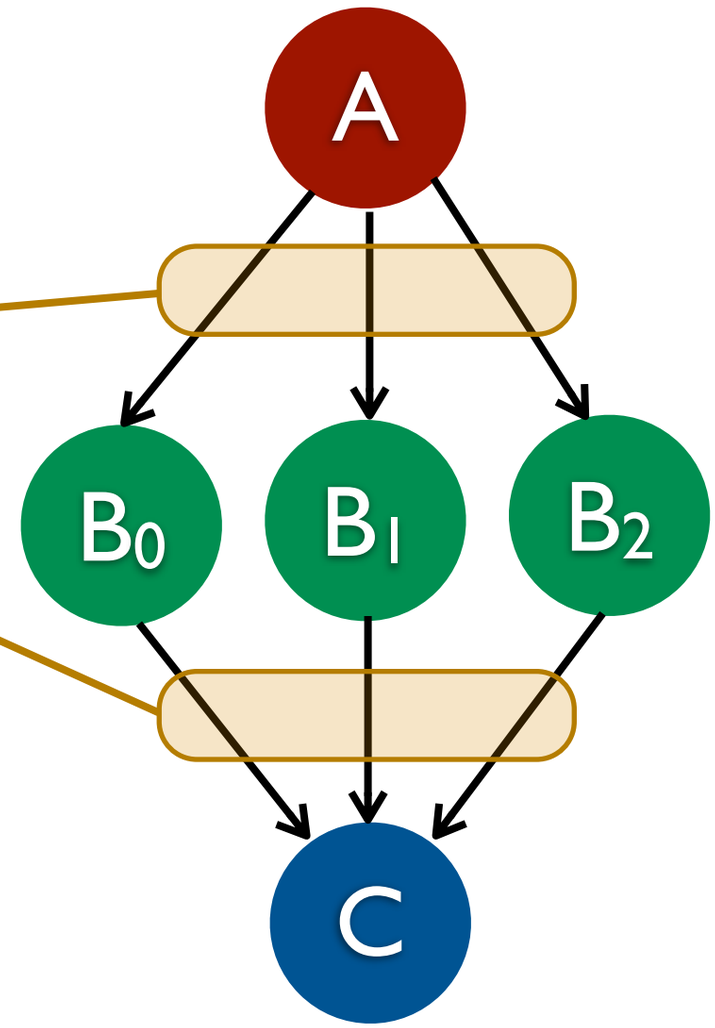
Ordering in Microsoft TPL

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

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

/*C*/
```



Ordering in Microsoft TPL

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

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

/*C*/
```



Sources of Task-Ordering

- 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-Ordering

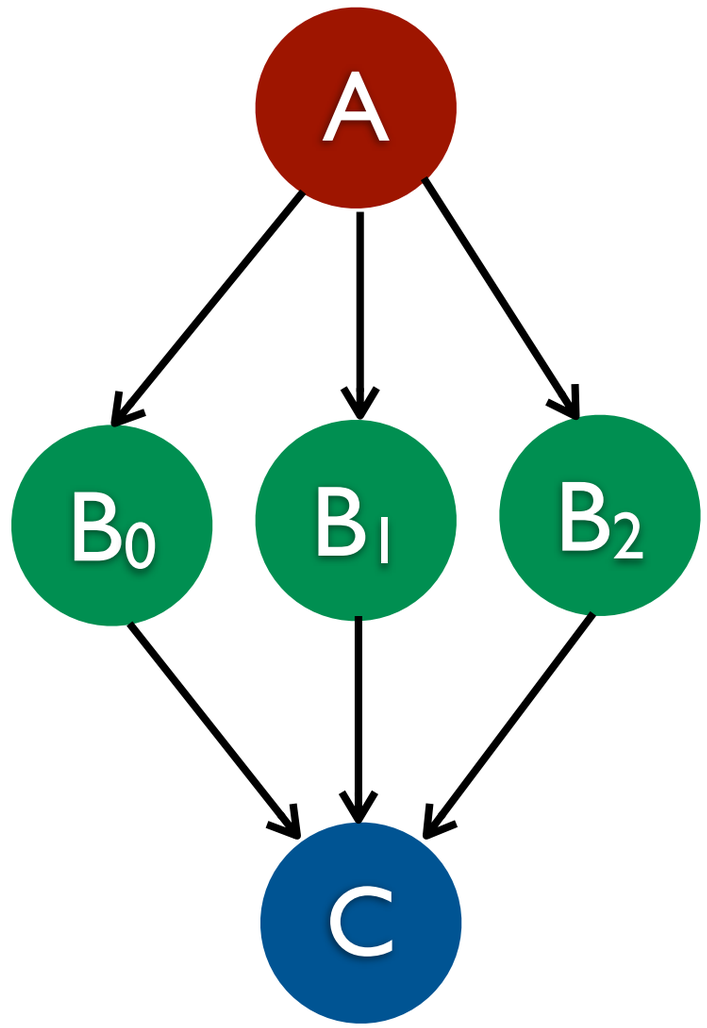
- Threads (Java)
  - Low-level synchronization, difficult to analyze
- Fork/join
  - Lexical
- Task Lib (TPL)
  - Feature explicit task ordering
  - Not much previous work here

Abstraction:

Tasks with explicit happens-before relationships

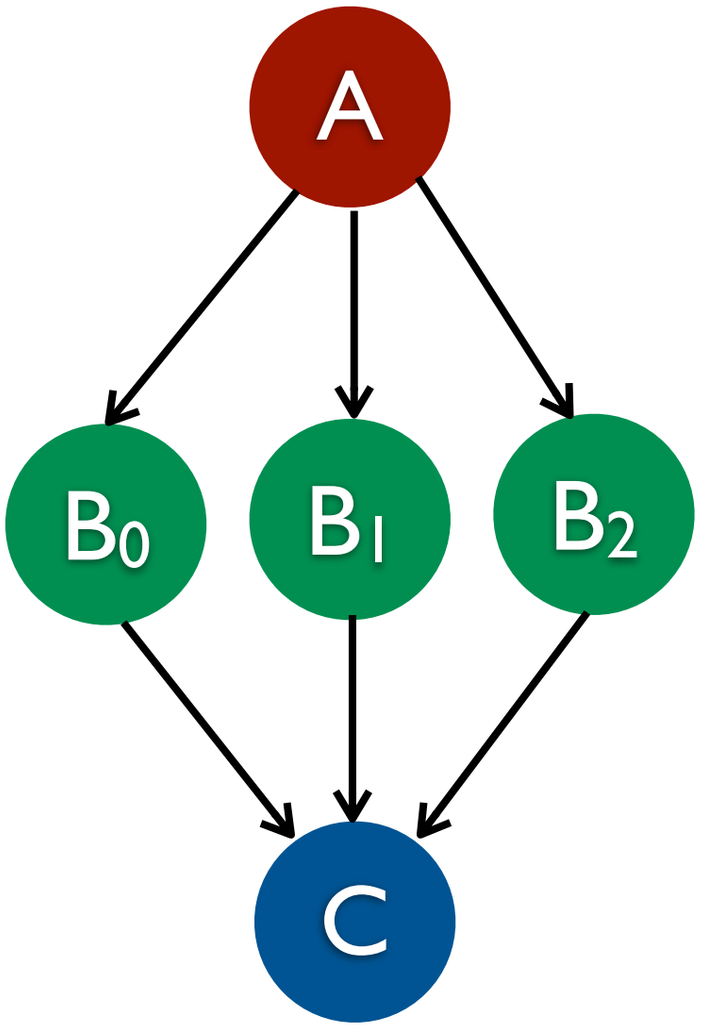
Our Model: Explicit Scheduling

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

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



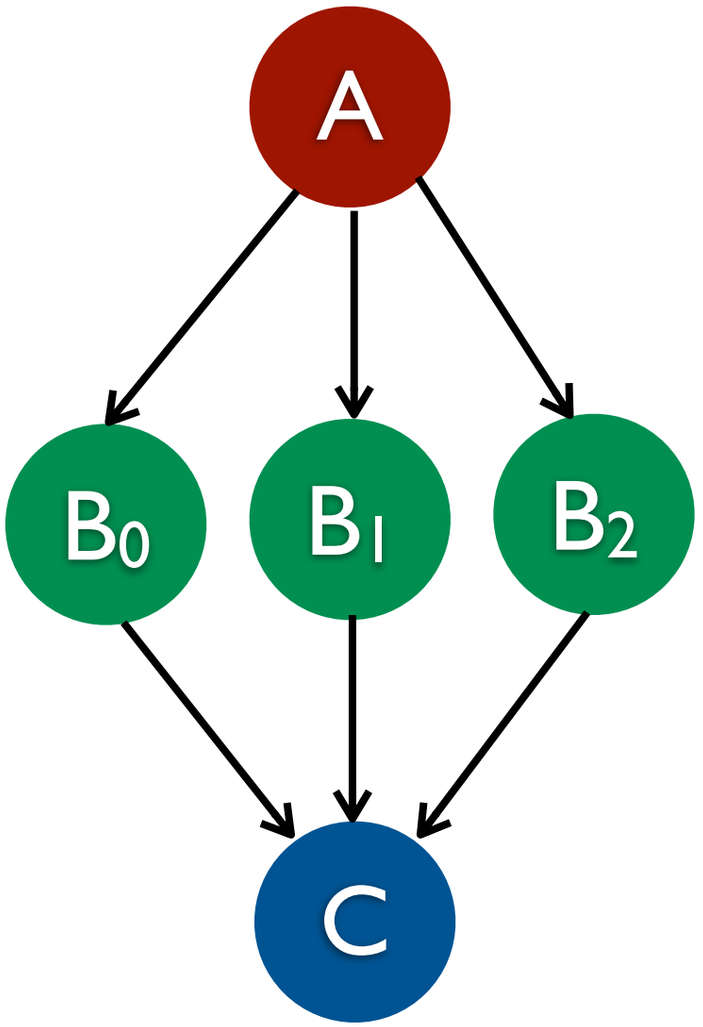
Our Model: Explicit Scheduling

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

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



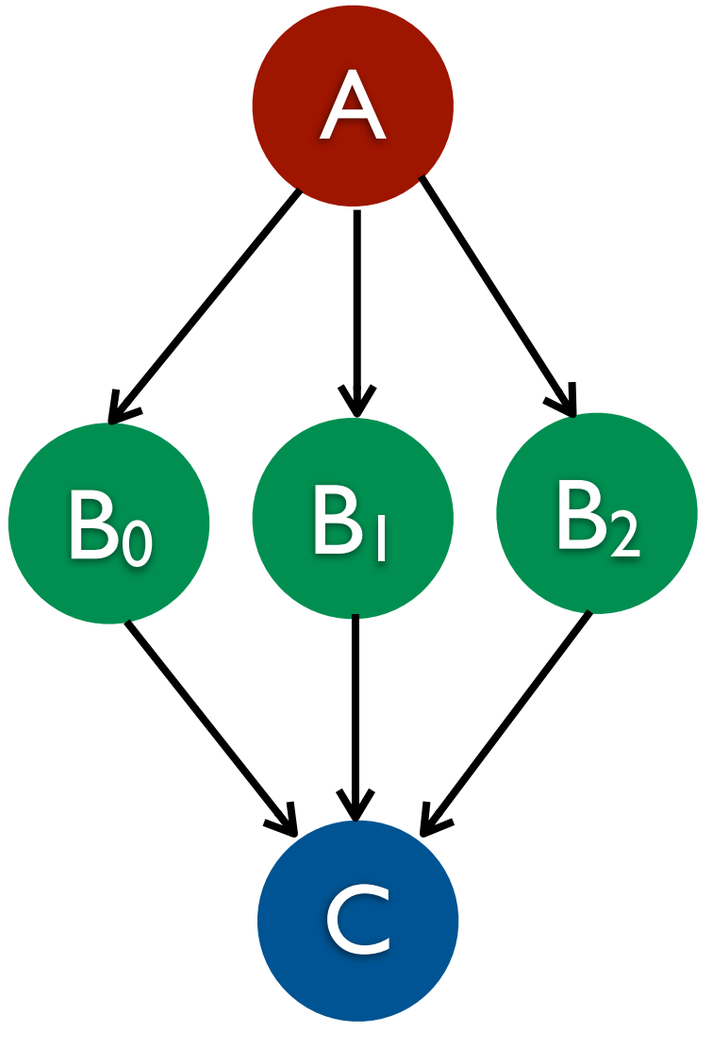
Our Model: Explicit Scheduling

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

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



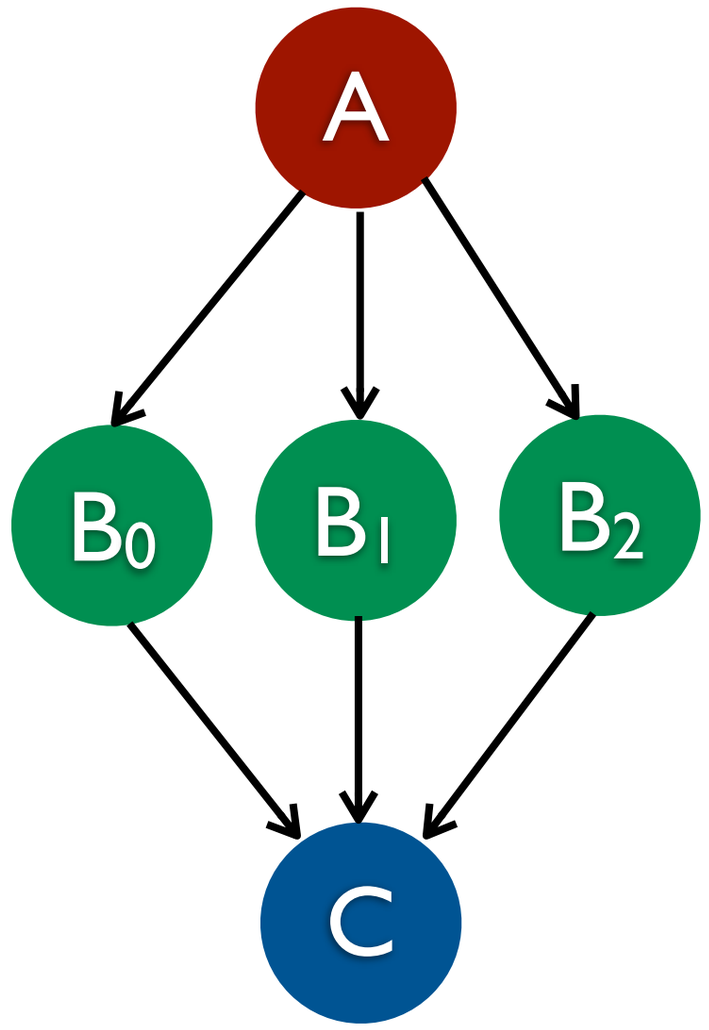
Our Model: Explicit Scheduling

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

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



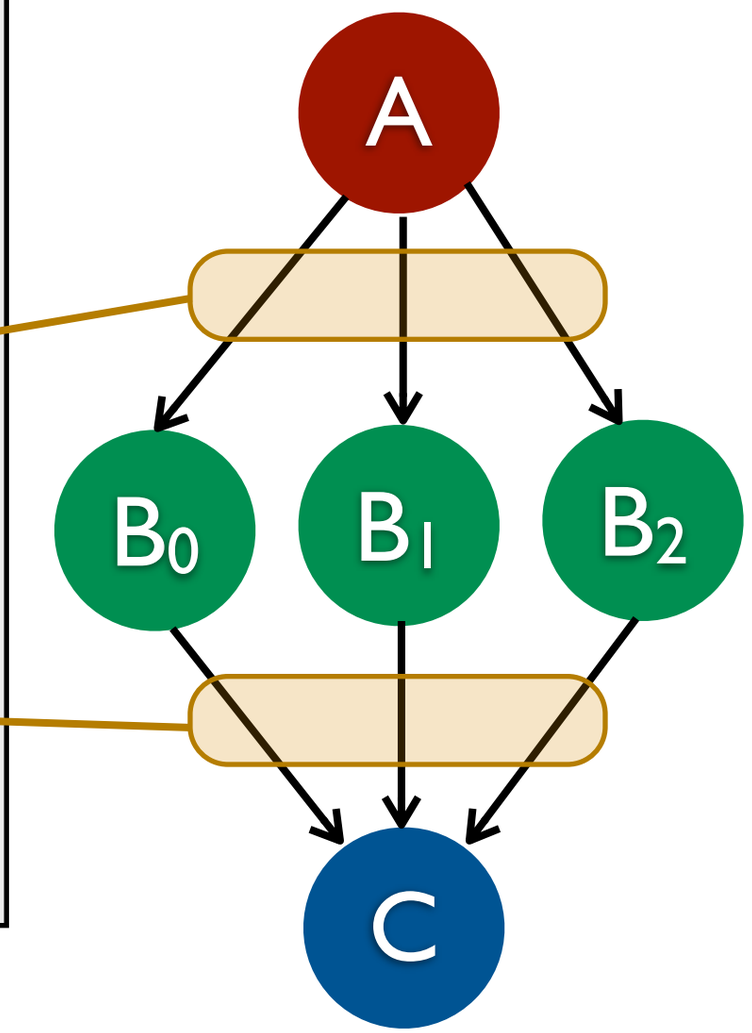
Our Model: Explicit Scheduling

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

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



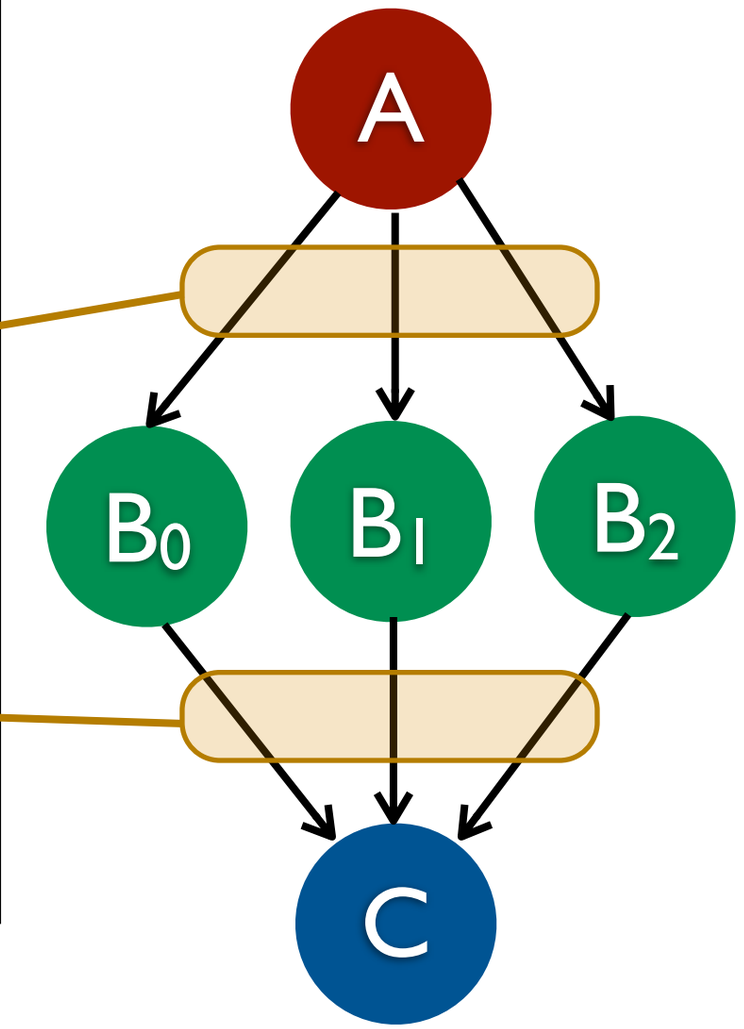
Our Model: Explicit Scheduling

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

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



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, ...

Outline

- Motivation
- Task Model
- Schedule Analysis
- Two Case Studies
- Conclusion

Schedule Analysis

Schedule Analysis

- Static analysis, works in two phases:

Schedule Analysis

- Static analysis, works in two phases:

1. Extract schedule summaries from bytecode

⇒ Task scheduling plus loop-sensitive ordering

Schedule Analysis

- Static analysis, works in two phases:

1. Extract schedule summaries from bytecode
   ⇒ Task scheduling plus loop-sensitive ordering

2. Until fixed-point reached:

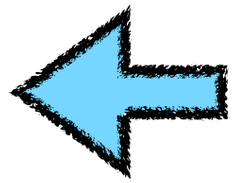
   Find tasks that may be created without ordering

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}
```

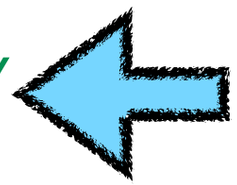
1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}
```



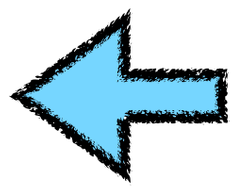
1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}
```



1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}
```



1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules   and
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules   and
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules   and
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and B*
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and B*
  with

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and B*
  with A ⇸ B

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and B*
  with A ⇸ B

task T2:
  schedules   and
```

Schedule Summary

1: Extract Summaries

```
task T1() {
  Task a = schedule /*A*/
  for(...) {
    Task b = schedule /*B*/
    a → b;
  }
}

task T2() {
  schedule /*A*/
  schedule /*B*/
}
```

```
task T1:
  schedules A and B*
  with A ⇸ B

task T2:
  schedules A and B
```

Schedule Summary

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B


task T2:
  schedules A and B
```

2: Find Unordered Tasks

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

```
task T1:
  schedules A and B*
  with A ⫫ B

task T2:
  schedules A and B
```

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B


task T2:
  schedules A and B
```

Ordered-ness:

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⊬ B


task T2:
  schedules A and B
```

Ordered-ness:

for T1: A ⊬ B

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B


task T2:
  schedules A and B
```

Ordered-ness:

for T1: A ⫫ B
for T2: ∅

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B


task T2:
  schedules A and B
```

Ordered-ness:

for T1: ~~A ⫫ B~~

for T2: ∅

not monotonic

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B

task T2:
  schedules A and B
```

Ordered-ness:

for T1: ~~A ⫫ B~~

for T2: ∅

not monotonic

Unordered-ness:

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B

task T2:
  schedules A and B
```

Ordered-ness:

for T1: ~~A ⫫ B~~

for T2: ∅

not monotonic

Unordered-ness:

for T1: B ‖ B

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B

task T2:
  schedules A and B
```

Ordered-ness:

for T1: ~~A ⫫ B~~

for T2: ∅

not monotonic

Unordered-ness:

for T1: B ‖ B

for T2: A ‖ B

2: Find Unordered Tasks

```
task T1:
  schedules A and B*
  with A ⫫ B

task T2:
  schedules A and B
```

Ordered-ness:

for T1: ~~A ⫫ B~~

for T2: ∅

not monotonic

Unordered-ness:

for T1: B ‖ B

for T2: A ‖ B

monotonic

Summary Schedule Analysis

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

Outline

- Motivation
- Task Model
- Schedule Analysis
- Two Case Studies
  - Sequentially Consistent Java
  - Dynamic Fractional Permissions
- Conclusion

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 instruction 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 instruction re-ordering

Inefficient without optimizations!

SC Example

Initially, x == y == 0

Task 1	Task 2
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

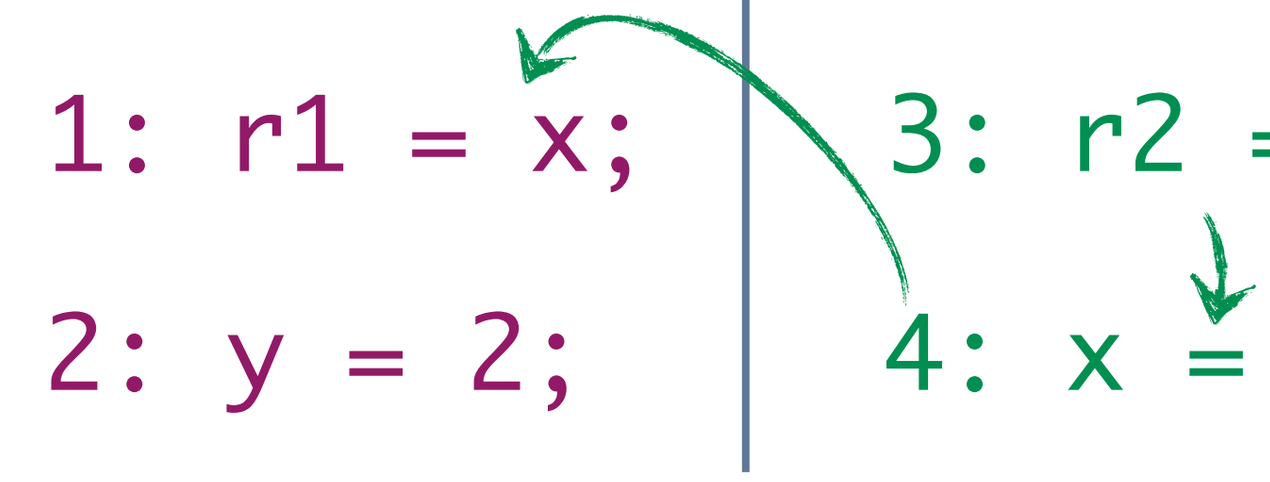
Task 1	Task 2
1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;

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

SC Example

Initially, x == y == 0

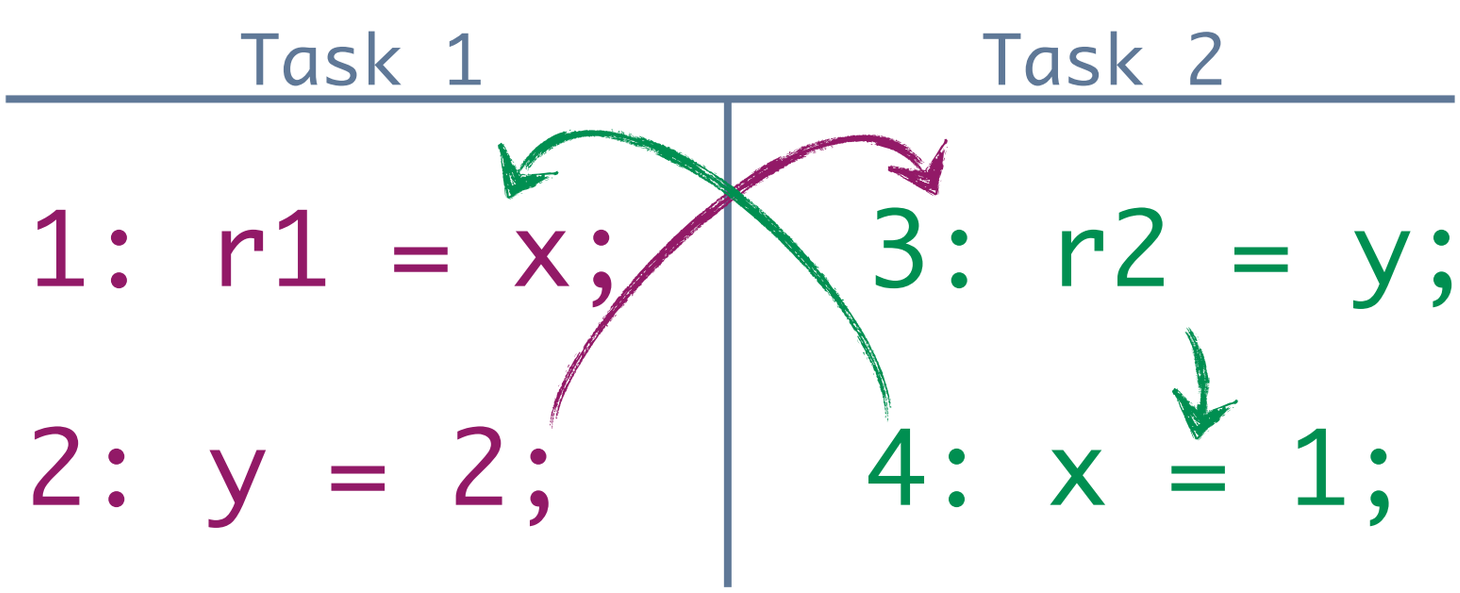
Task 1	Task 2
1: r1 = x;	3: r2 = y;
2: y = 2;	4: x = 1;



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

SC Example

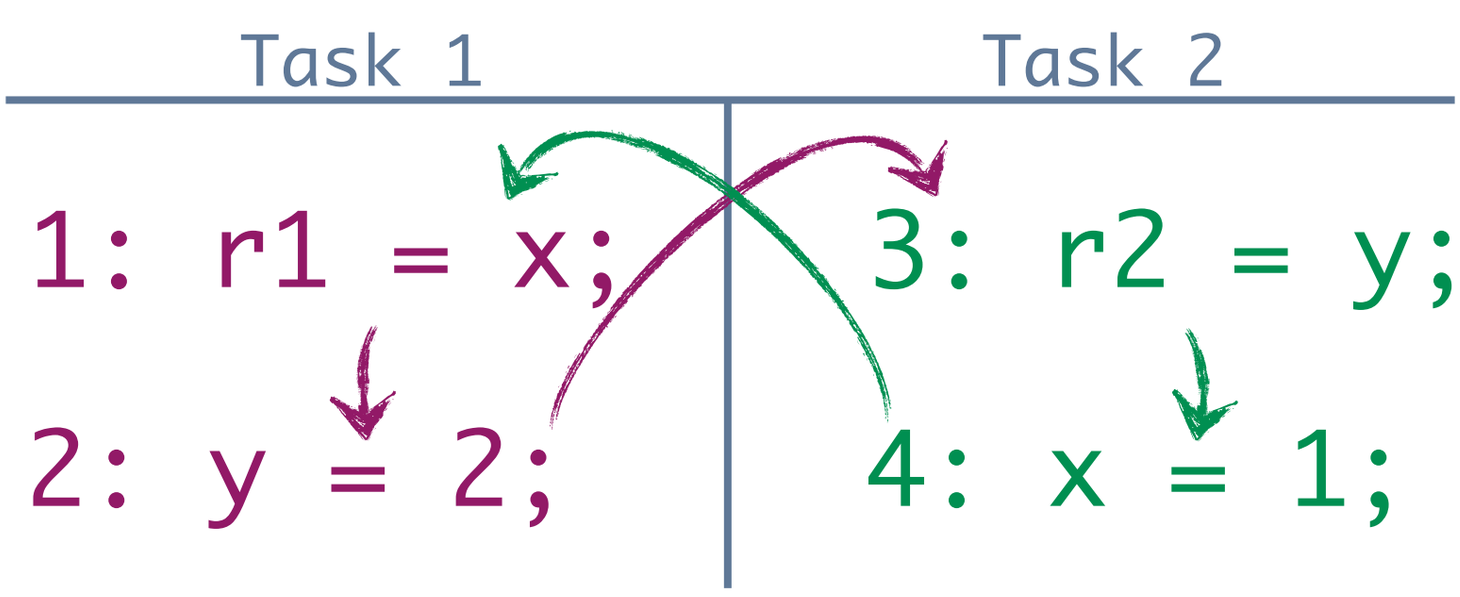
Initially, x == y == 0



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

SC Example

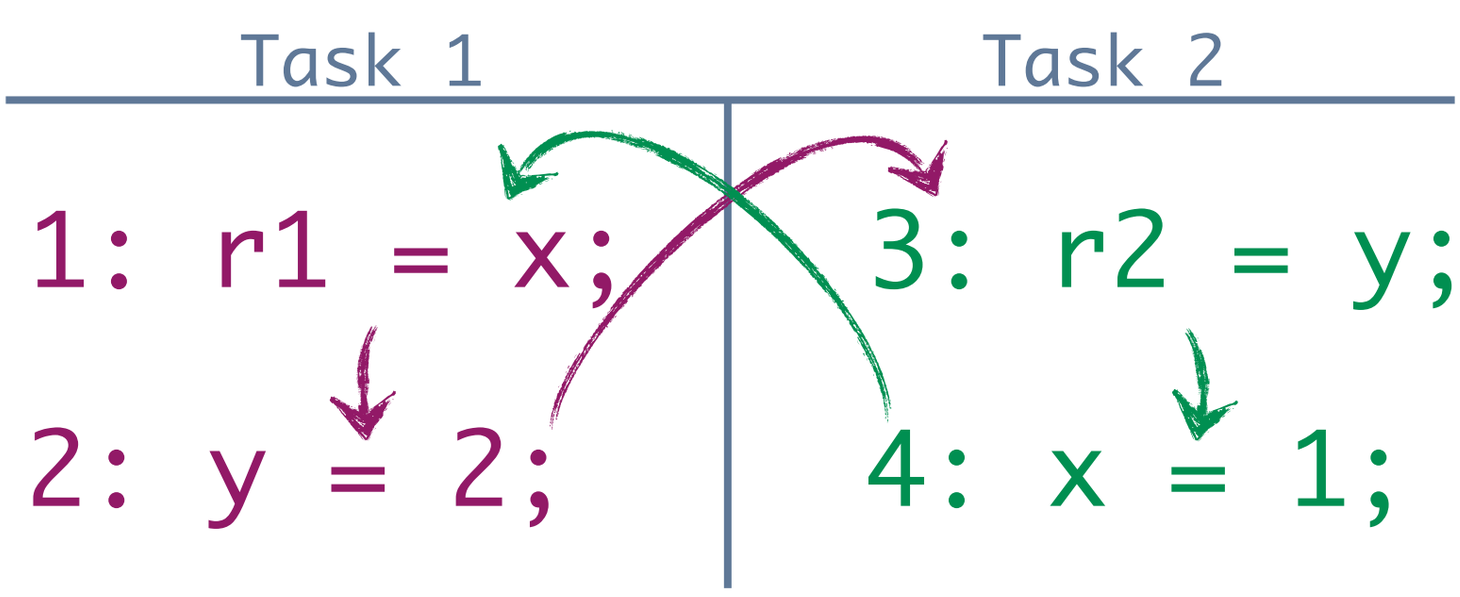
Initially, x == y == 0



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

SC Example

Initially, x == y == 0



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

No, if "sequentially consistent"

SC Example

Initially, x == y == 0

Task 1	Task 2
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 not needed
  - Different objects ∨ ordered tasks

Evaluation Setup

- Intel Core 2 Duo, 2.8 GHz, 4Gb RAM

  ⇒2 Java Threads

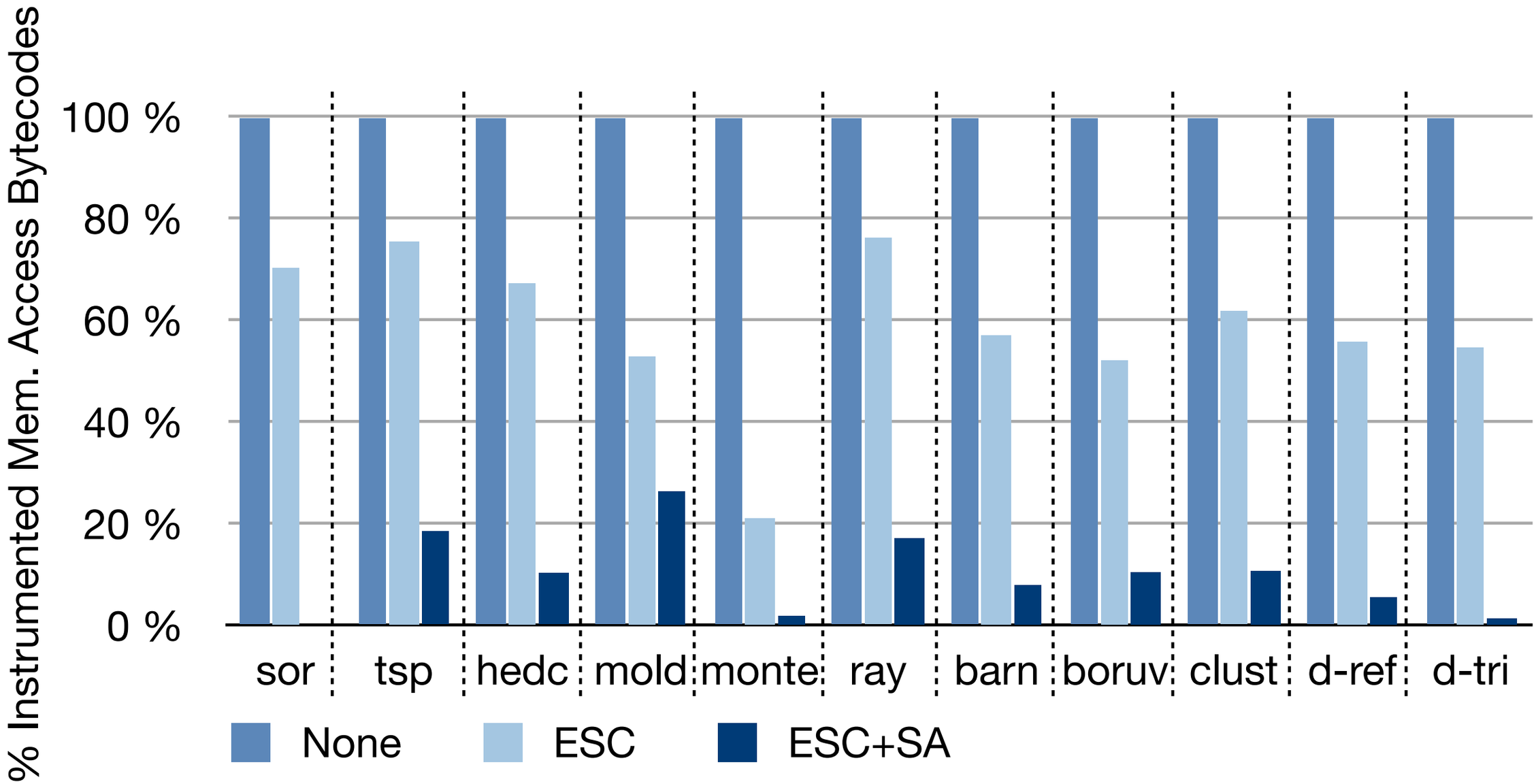
- Java 1.6.0 (Mac)

- Benchmarks
  - Java Grande: numeric apps
  - Erco Benchmarks: OO-style parallelism
  - Lonestar: larger applications, irregular behavior

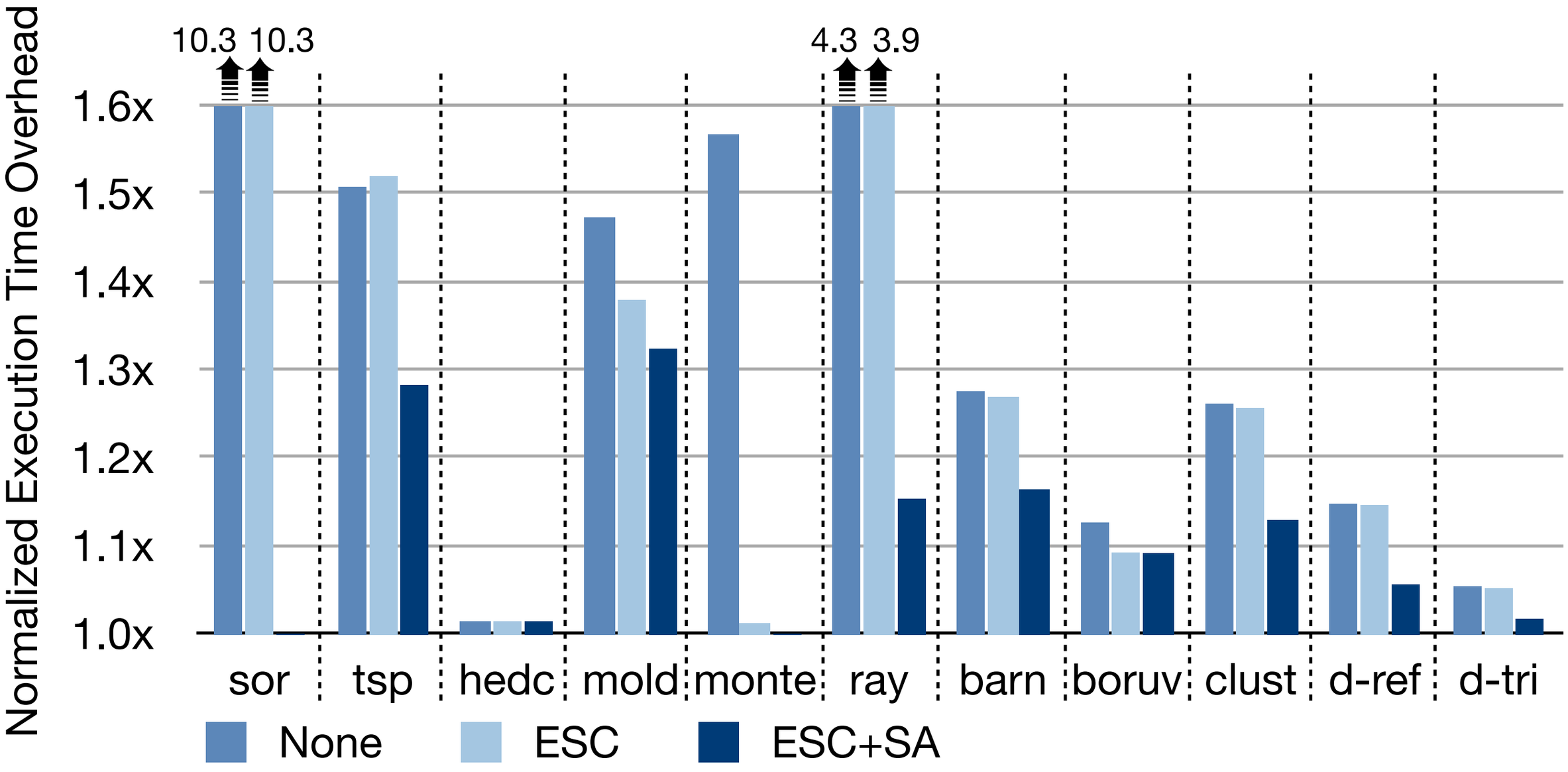
Configurations

- Relaxed memory model, programmer-provided SC (`Baseline`)
- SC, no optimizations (None)
- SC, escape analysis (ESC)
- SC, schedule analysis + escape analysis (SA+ESC)

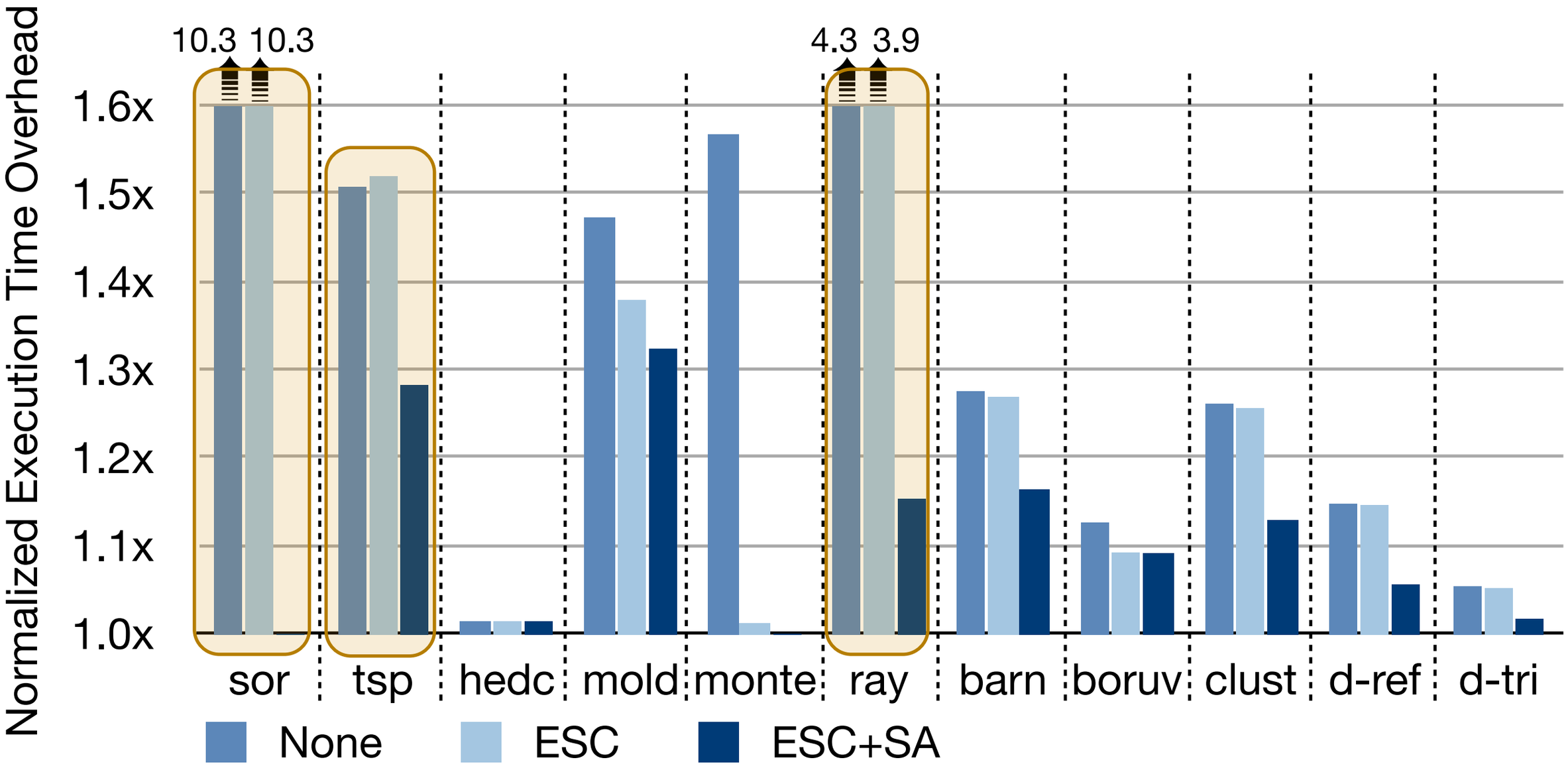
Instrumentation Overhead



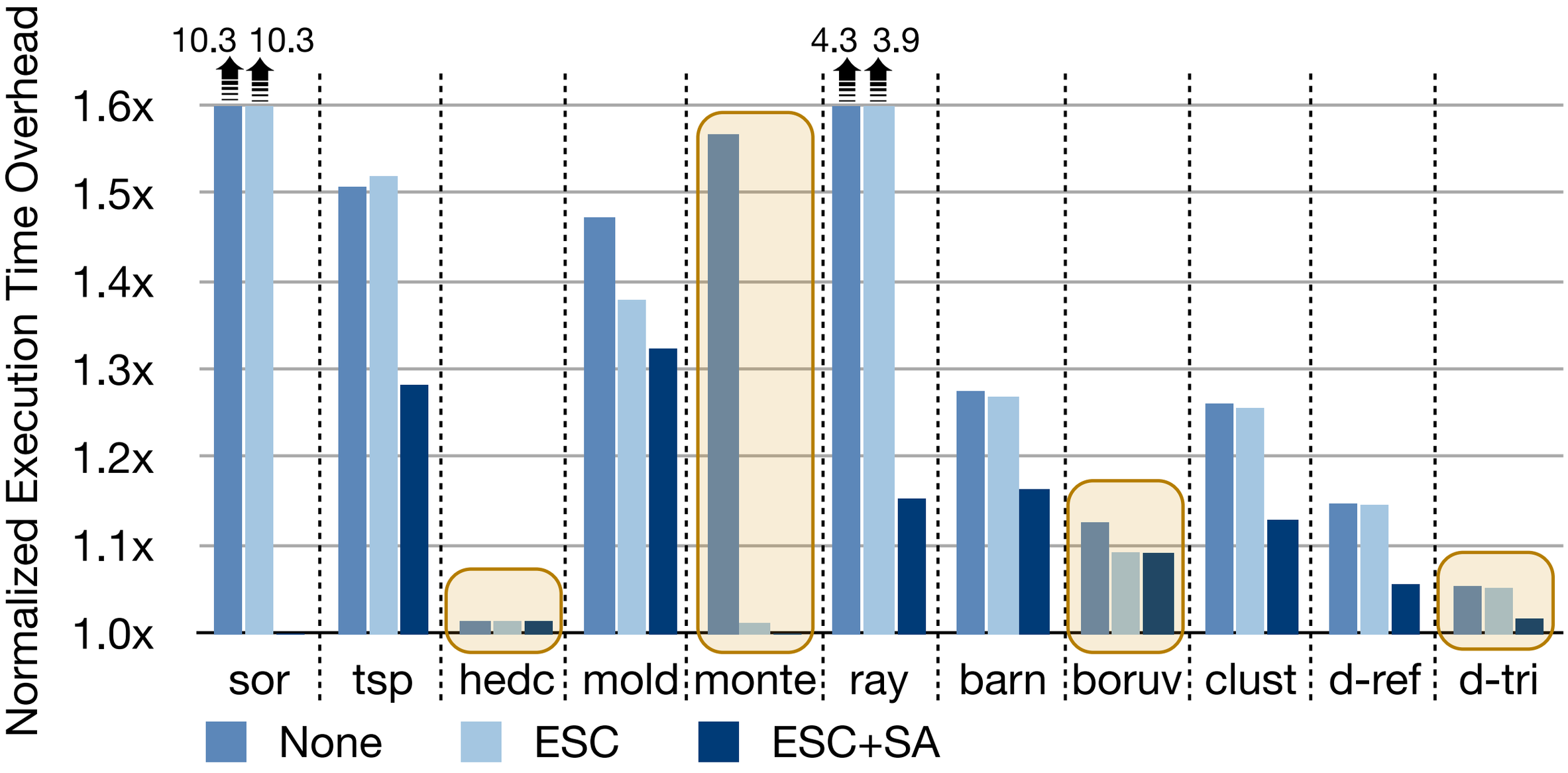
Execution Time



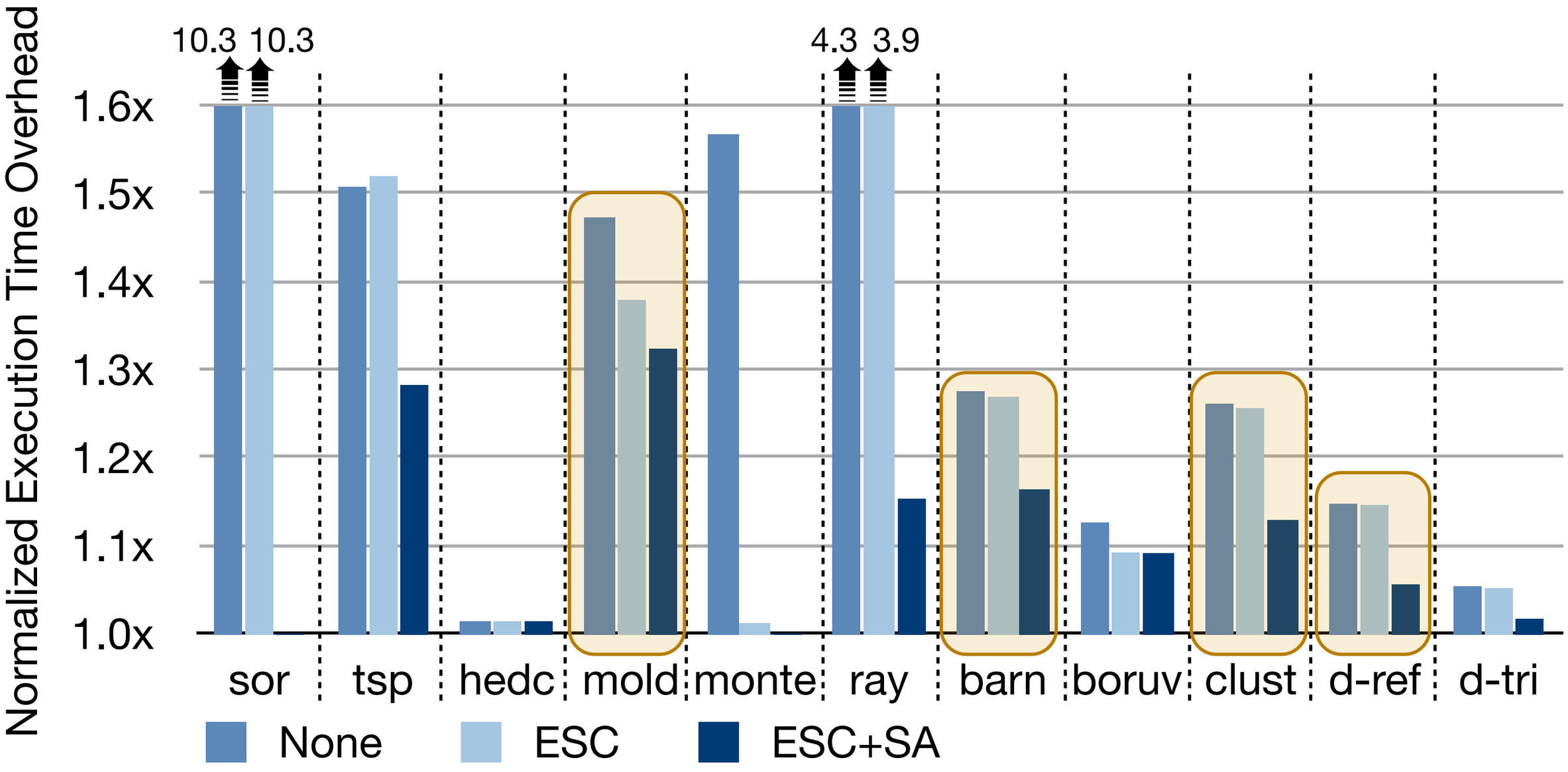
Execution Time



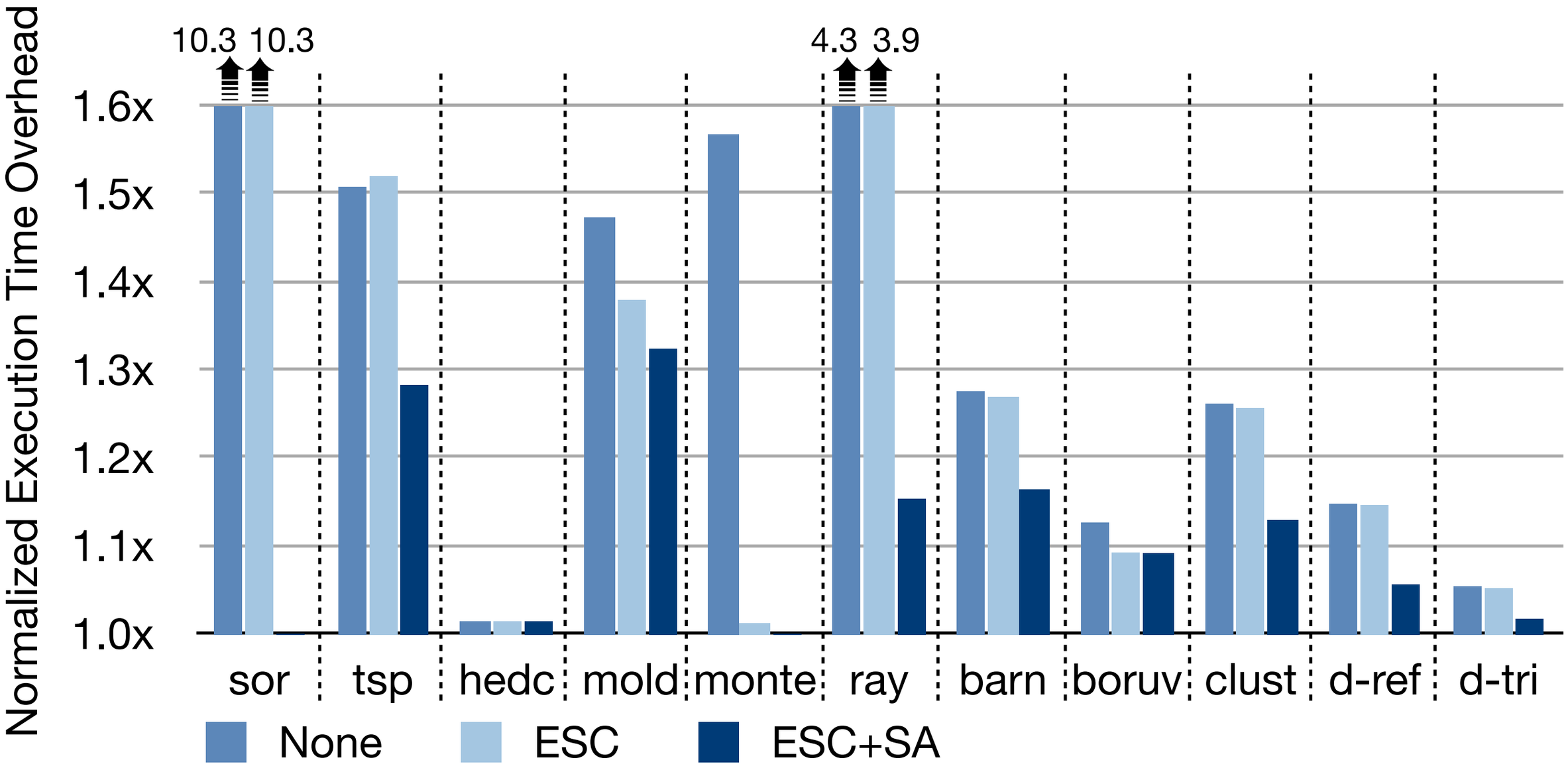
Execution Time



Execution Time



Execution Time



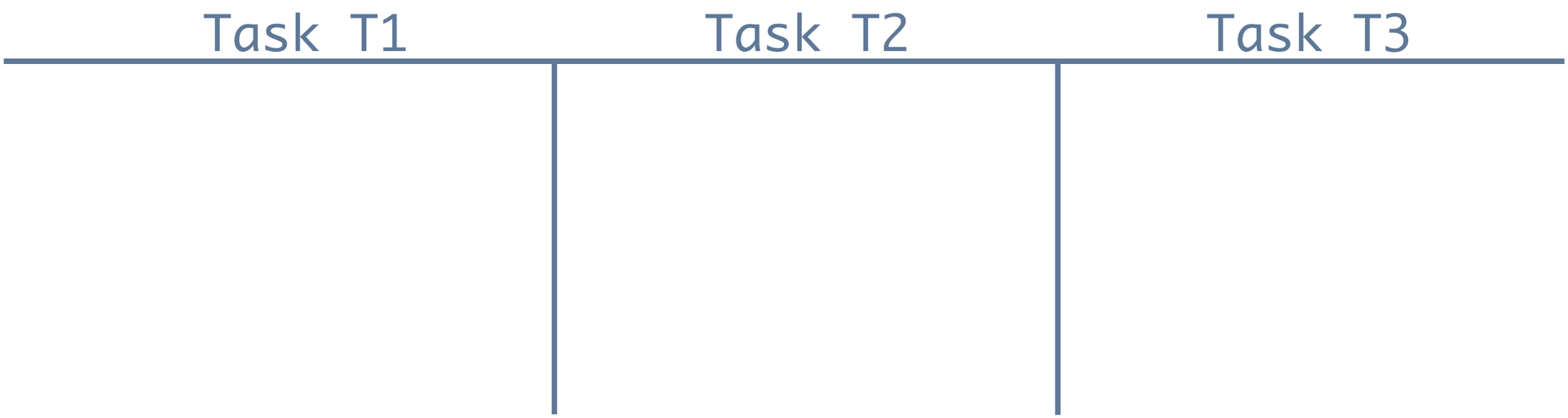
Outline

- Motivation
- Task Model
- Schedule Analysis
- Two Case Studies
  - Sequentially Consistent Java
  - Dynamic Fractional Permissions
- Conclusion

Dynamic Fractional Permissions

- Dynamic version of static data-race detection framework [Boyland 2003]
- Programmer manages access permissions:
  - Every object o has associated access control list
  - `Task T ∈ ACL(o) ⇒ read access`
  - `Task T only task in ACL(o) ⇒ write access`
- Compiler inserts permission checks

Permission Operations



Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();`		

Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();`		

ACL(x): {T1}

Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();`		

ACL(x): {T1}

Permissions:	T1	T2	T3
Can read x:	✓	✗	✗
Can write x:	✓	✗	✗

Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();` `x.Add(T2);`		

ACL(x): {T1}

Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();` `x.Add(T2);`		

ACL(x): {T1, T2}

Permission Operations

Task T1	Task T2	Task T3
`x = new Obj();` `x.Add(T2);`		

ACL(x): {T1, T2}

Permissions:	T1	T2	T3
Can read x:	✓	✓	✗
Can write x:	✗	✗	✗

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`		

ACL(x): {T1, T2}

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`		

ACL(x): {T3, T2}

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`		

ACL(x): {T3, T2}

Permissions:	T1	T2	T3
Can read x:	✗	✓	✓
Can write x:	✗	✗	✗

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`
	`x.Yield(T3);`	

ACL(x): {T3, T2}

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`
	`x.Yield(T3);`	

ACL(x): {T3}

Permission Operations

Task T1	Task T2	Task T3

`x = new Obj();`
`x.Add(T2);`
`x.Yield(T3);`
	`x.Yield(T3);`	

ACL(x): {T3}

Permissions:	T1	T2	T3
Can read x:	✗	✗	✓
Can write x:	✗	✗	✓

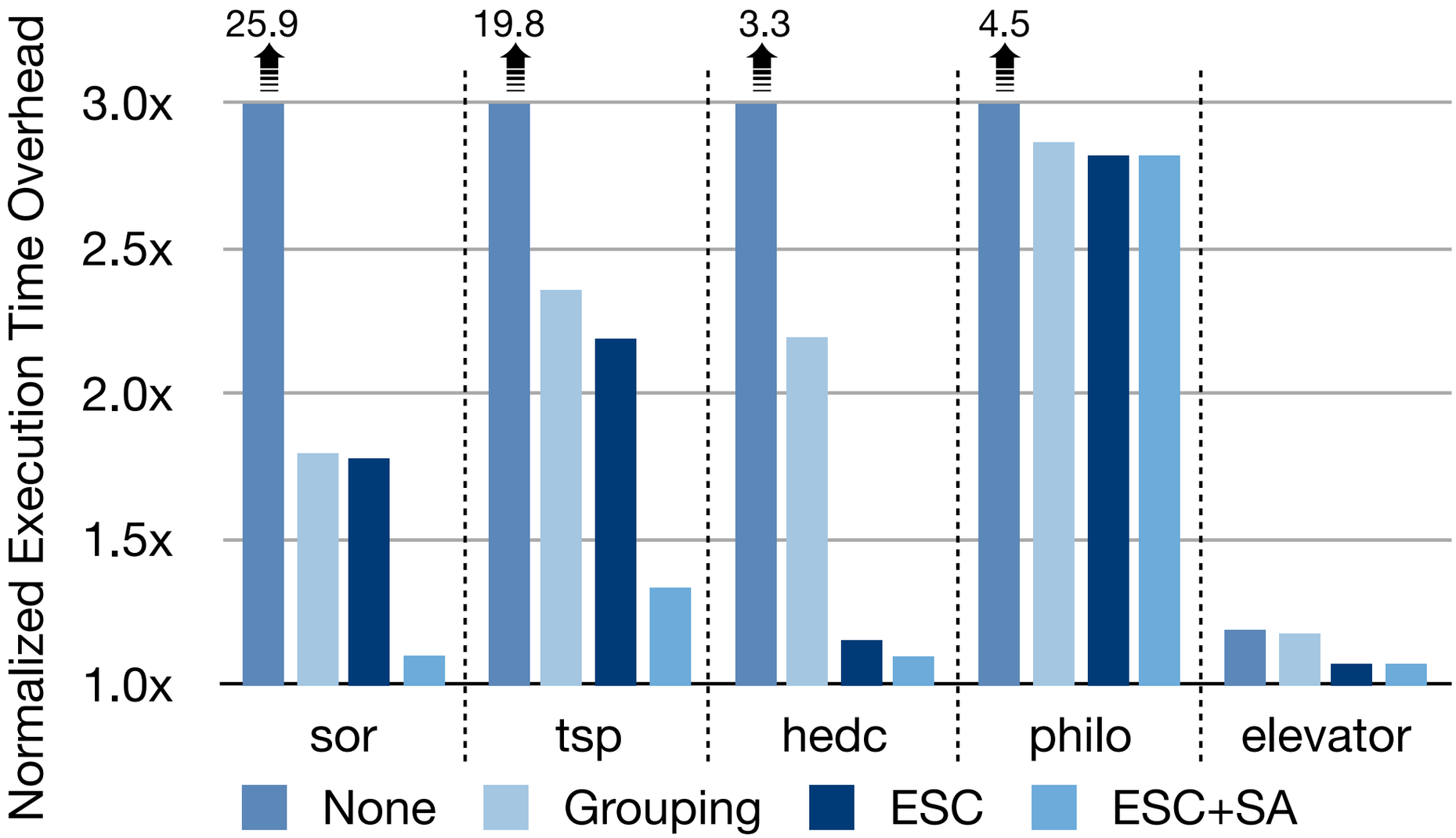
Optimizations

- Group redundant permission checks (method-local)
- Remove `new`, `Add`, and `Yield` operations for task-local objects
- Reduce number of inserted permission checks

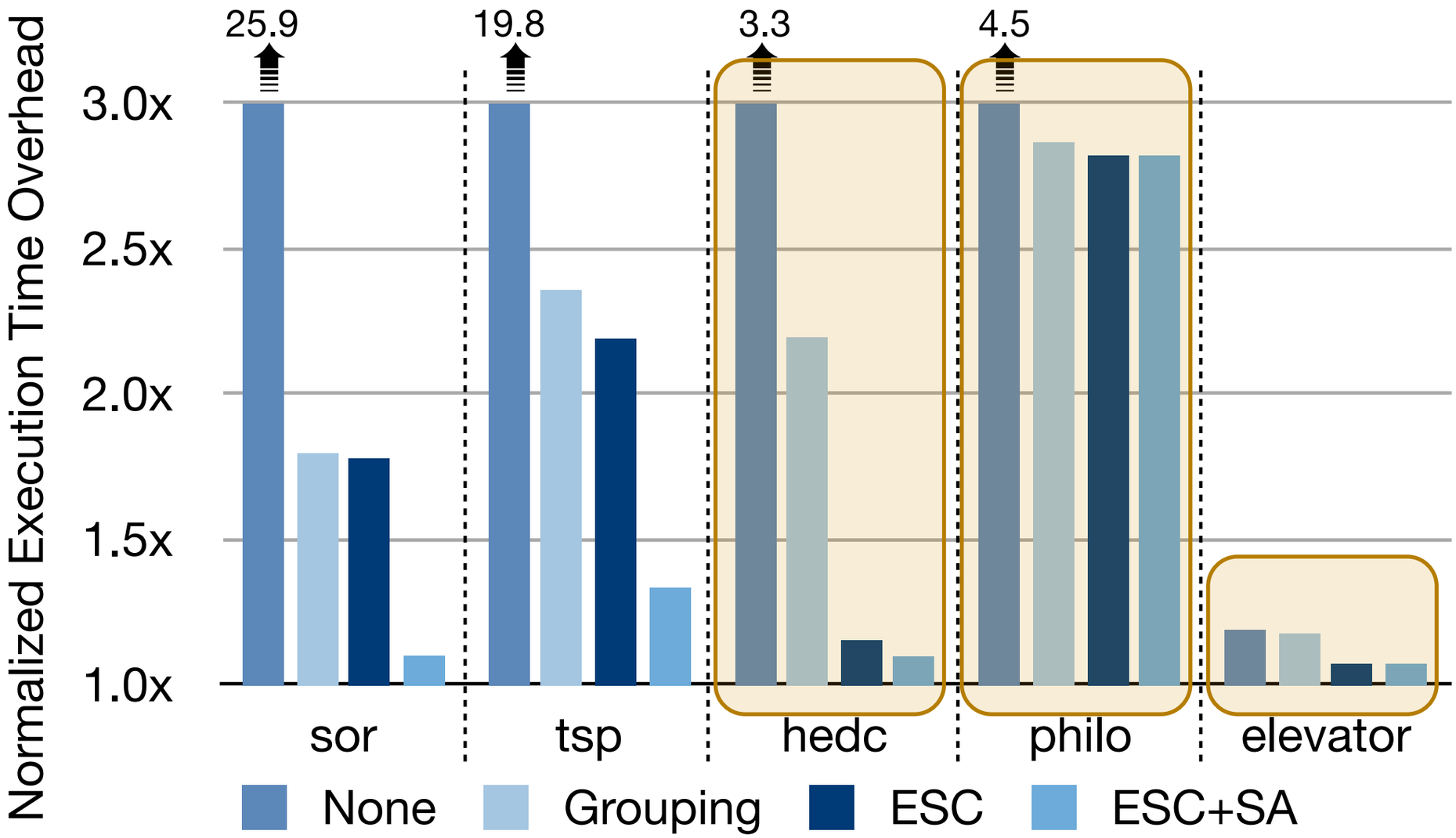
Configurations

- No permission system (`Baseline`)
- No optimizations, permission checks on all accesses (None)
- Grouping of redundant permission checks (Group)
- Escape analysis (ESC)
- Schedule analysis + escape analysis (SA+ESC)

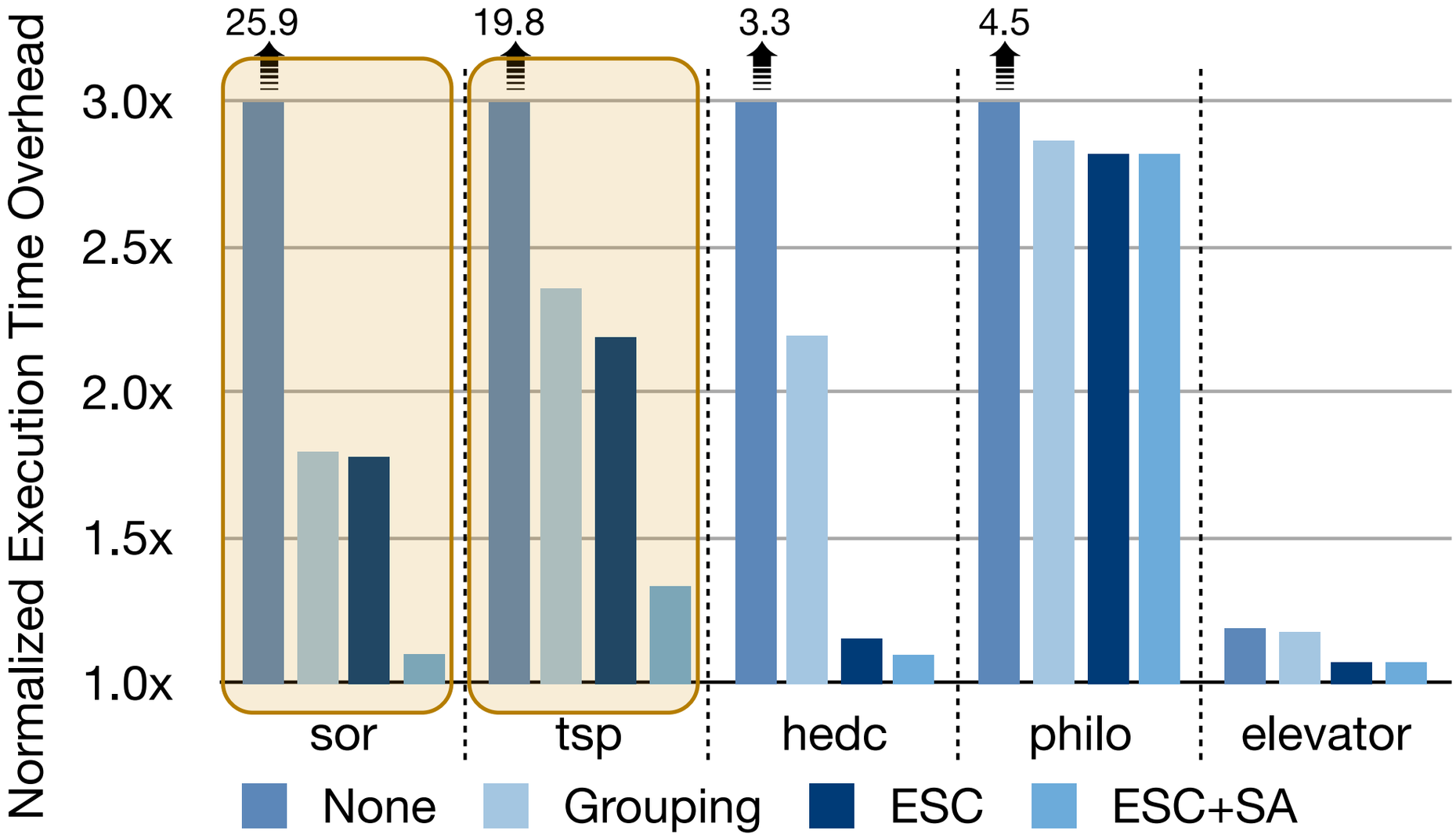
Execution Time



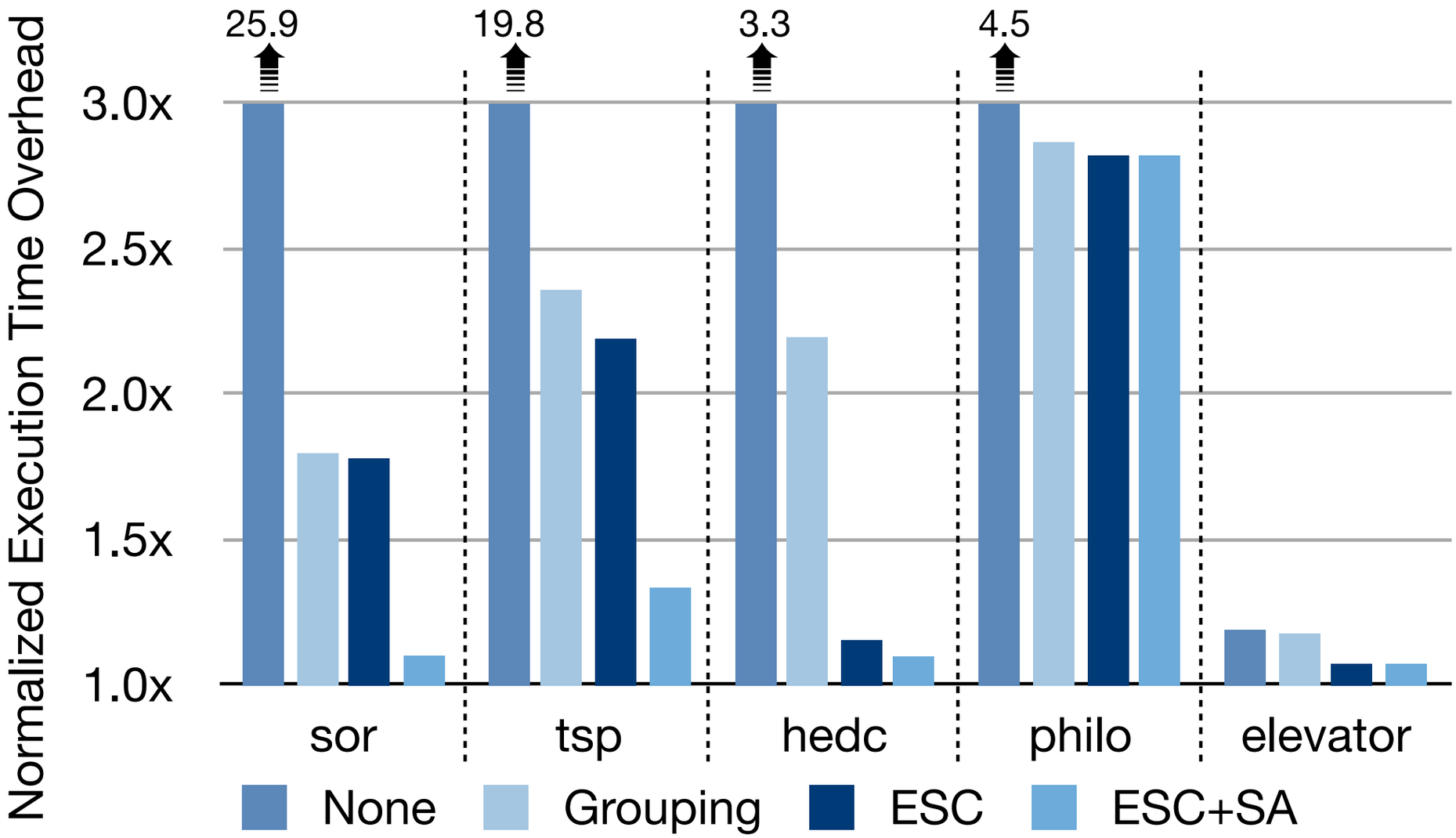
Execution Time



Execution Time



Execution Time



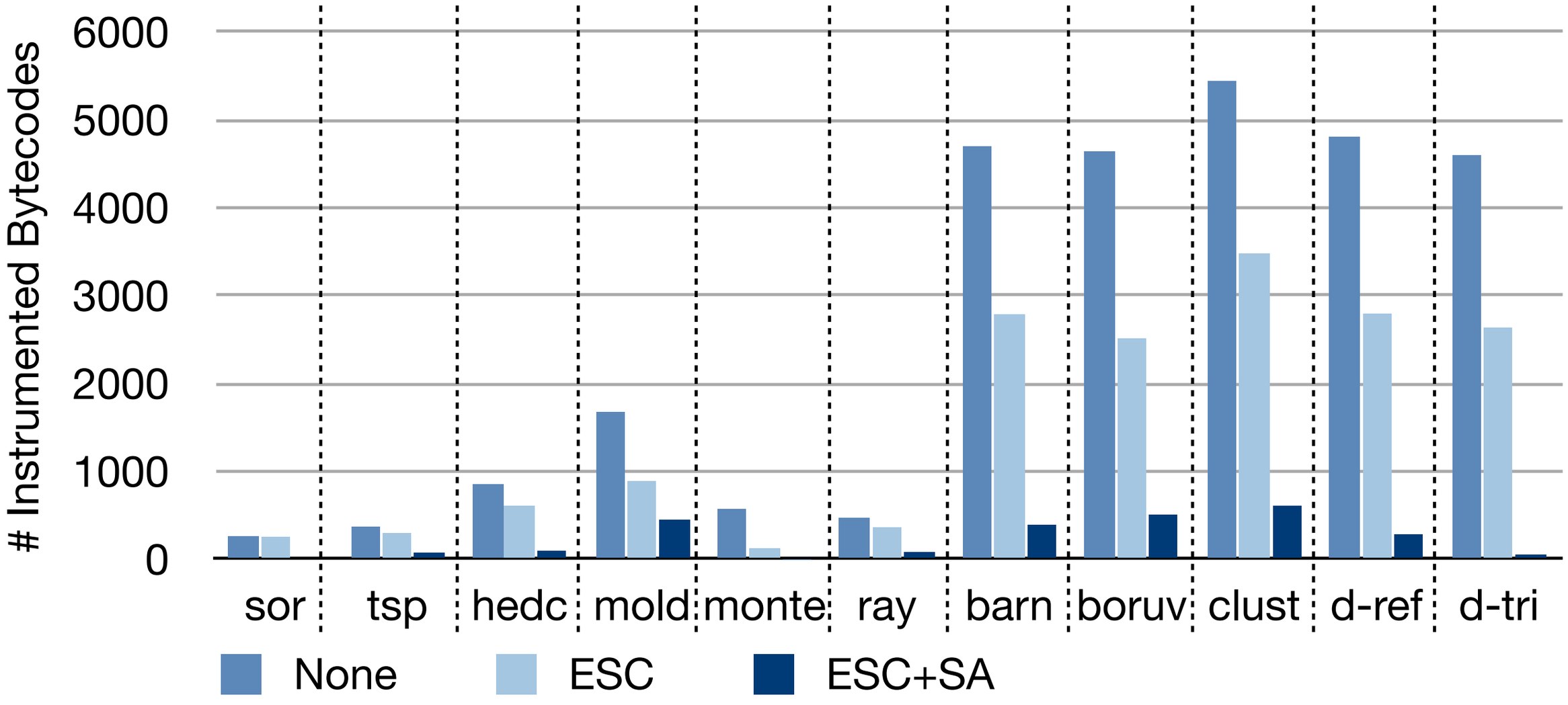
Conclusion

- Compilers for shared-memory parallel programs can exploit task-order information
- Schedule analysis:
  - can extract task-order information from real-world programs
  - provides basis for effective optimizations

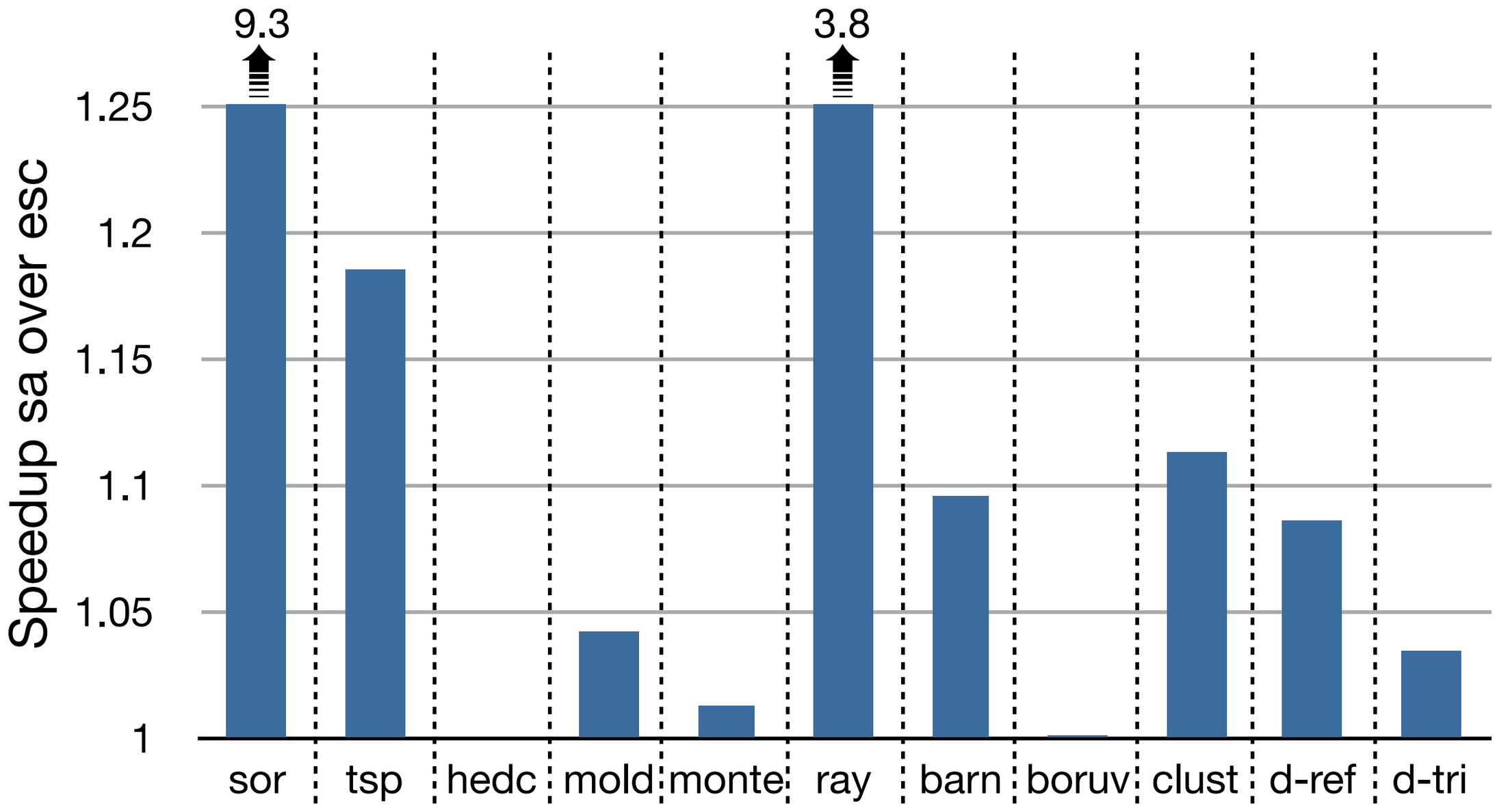
Implementation

- Bytecode-to-bytecode translation with `getVolatile()` and `putVolatile()` intrinsics
- Analyzes bytecode in SSA form
- Wala framework for analysis,
- Javassist for code generation

Instrumentation Overhead



Speedup SA+ESC over 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

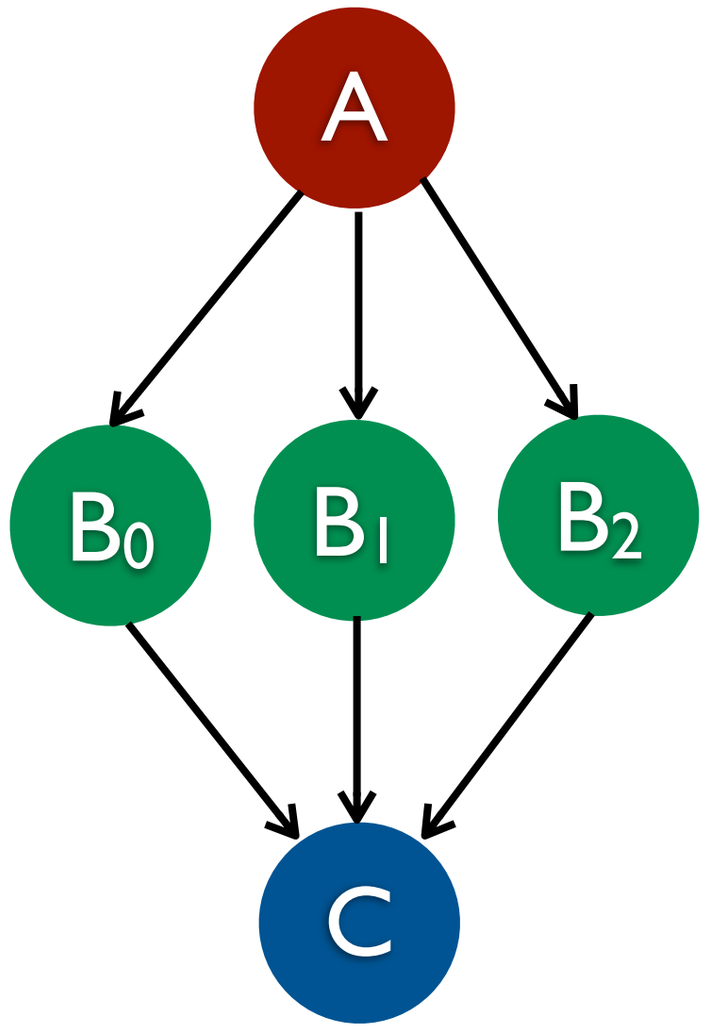
- **MHP Analysis.** (Agarwal et al)

Ordering with Threads

```
/*A*/

Thread[] ts = new Thread[3];
for(int i=0; i<3; i++) {
  ts[i] = new MyThread(/*B*/);
  ts[i].start();
}
for(int i=0; i<3; i++)
  ts[i].join();

/*C*/
```

Ordering with Threads

```
/*A*/

Thread[] ts = new Thread[3];
for(int i=0; i<3; i++) {
  ts[i] = new MyThread(/*B*/);
  ts[i].start();
}
for(int i=0; i<3; i++)
  ts[i].join();

/*C*/
```

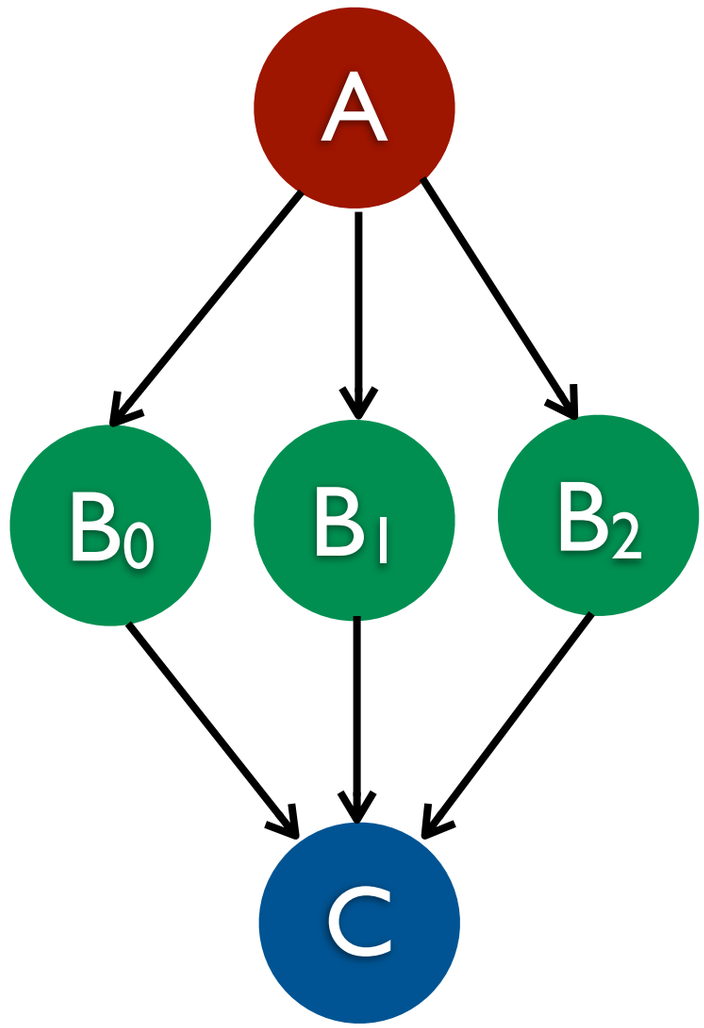


Ordering in OpenMP

```
/*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*/
```



Permission Operations

Permission Operations

```
/*permission management*/
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}

@T1: o.Add(T2)            ⇒ ACL(o) u {T2}
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}

@T1: o.Add(T2)            ⇒ ACL(o) u {T2}

@T1: o.Yield(T2)          ⇒ [T1↦T2]ACL(o)
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}

@T1: o.Add(T2)            ⇒ ACL(o) u {T2}

@T1: o.Yield(T2)          ⇒ [T1↦T2]ACL(o)


/*inserted by compiler*/
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}

@T1: o.Add(T2)            ⇒ ACL(o) u {T2}

@T1: o.Yield(T2)          ⇒ [T1↦T2]ACL(o)


/*inserted by compiler*/
@T1: o.CheckRead()        ⇒ true iff T1 ∈ ACL(o)
```

Permission Operations

```
/*permission management*/
@T1: o = new C()          ⇒ ACL(o) = {T1}

@T1: o.Add(T2)            ⇒ ACL(o) u {T2}

@T1: o.Yield(T2)          ⇒ [T1↦T2]ACL(o)


/*inserted by compiler*/
@T1: o.CheckRead()    ⇒ true iff T1 ∈ ACL(o)

@T1: o.CheckWrite()   ⇒ true iff ACL(o) == {T1}
```