Static Analysis of Dynamic Schedules

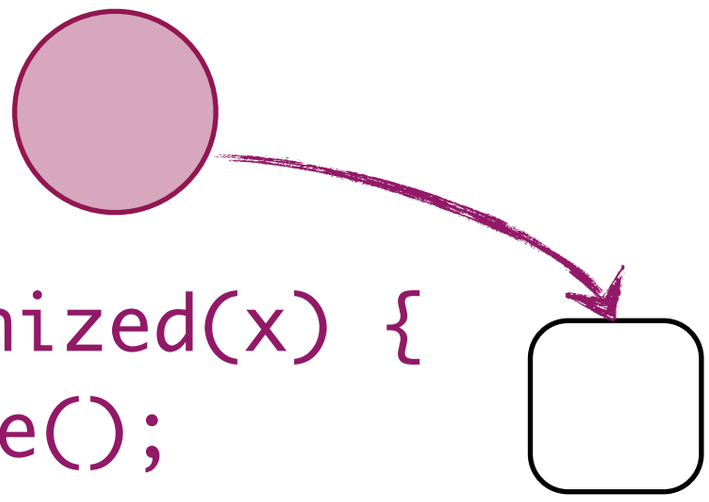
and its Application to Optimization of Parallel Programs

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

Example

Example

task A

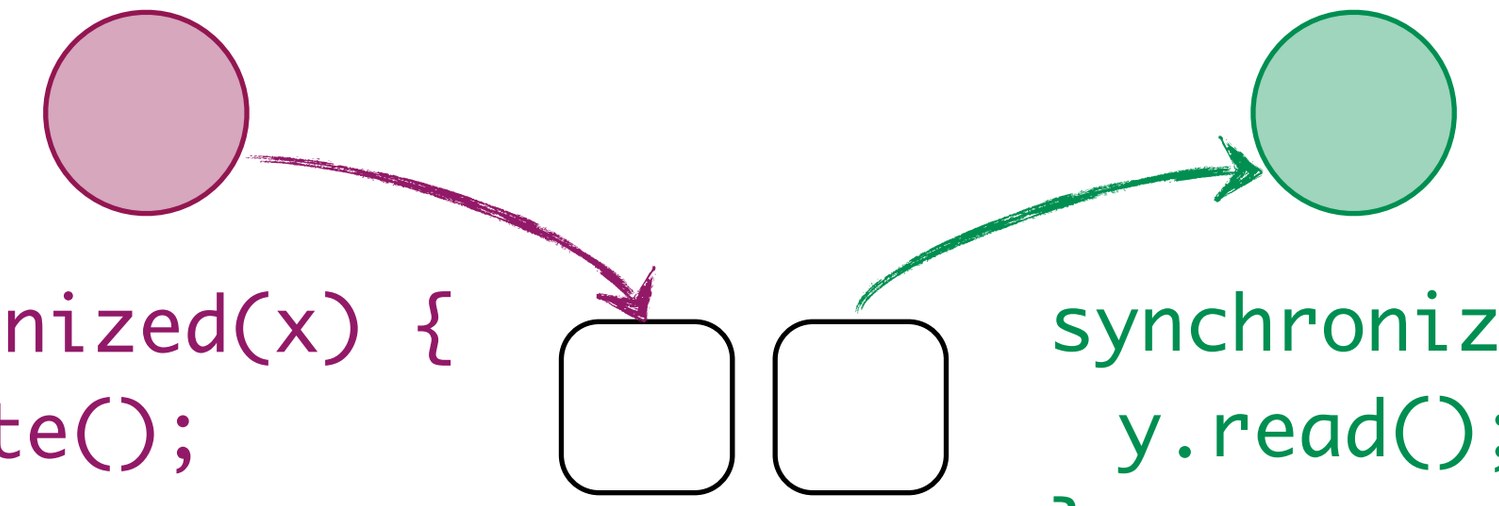


```
synchronized(x) {
  x.write();
}
```

Example

task A

task B

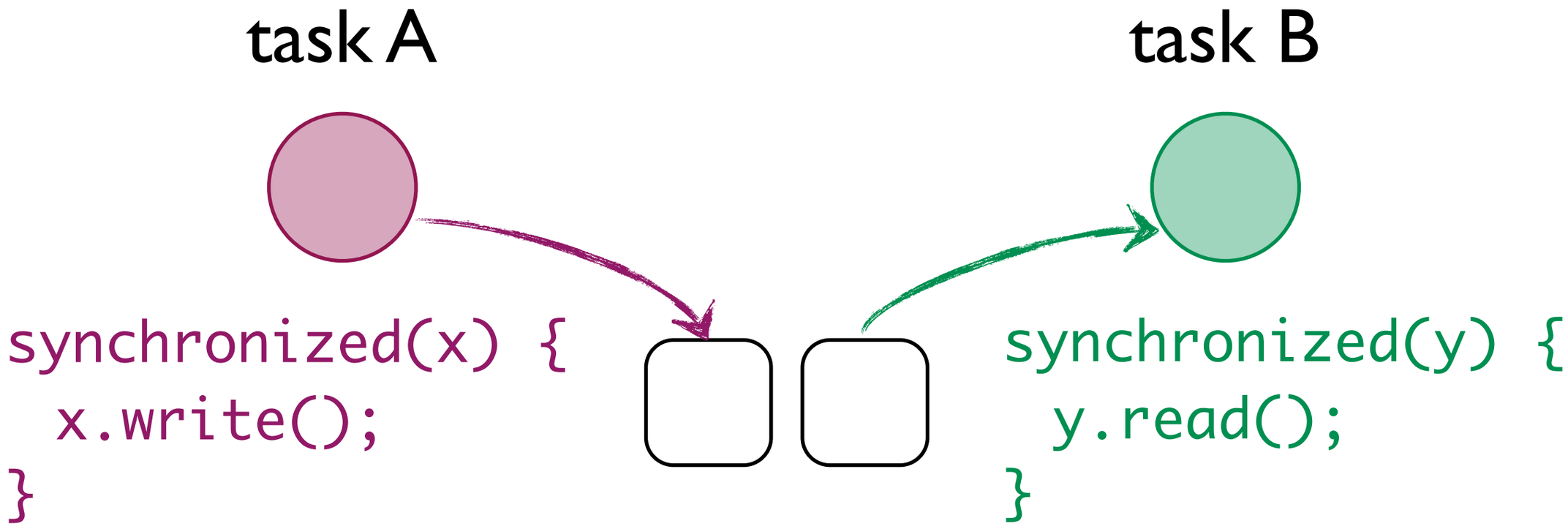


```
synchronized(x) {
  x.write();
}
```

```
synchronized(y) {
  y.read();
}
```

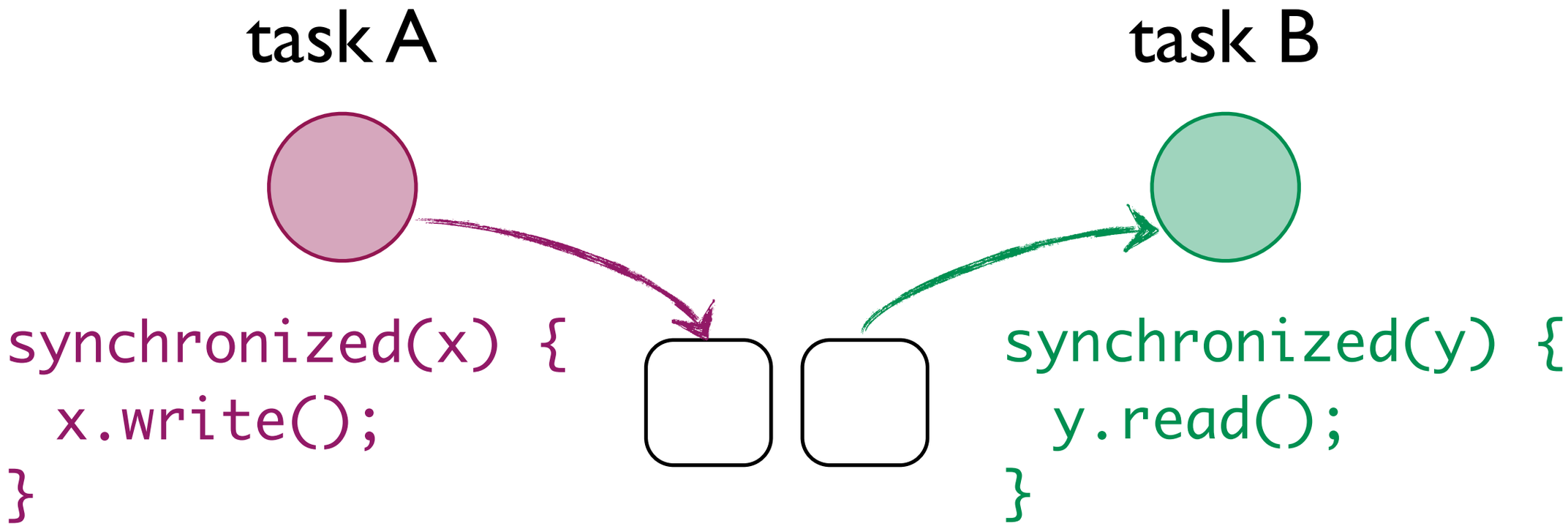
Example

- Can we remove synchronization for x, y?



Example

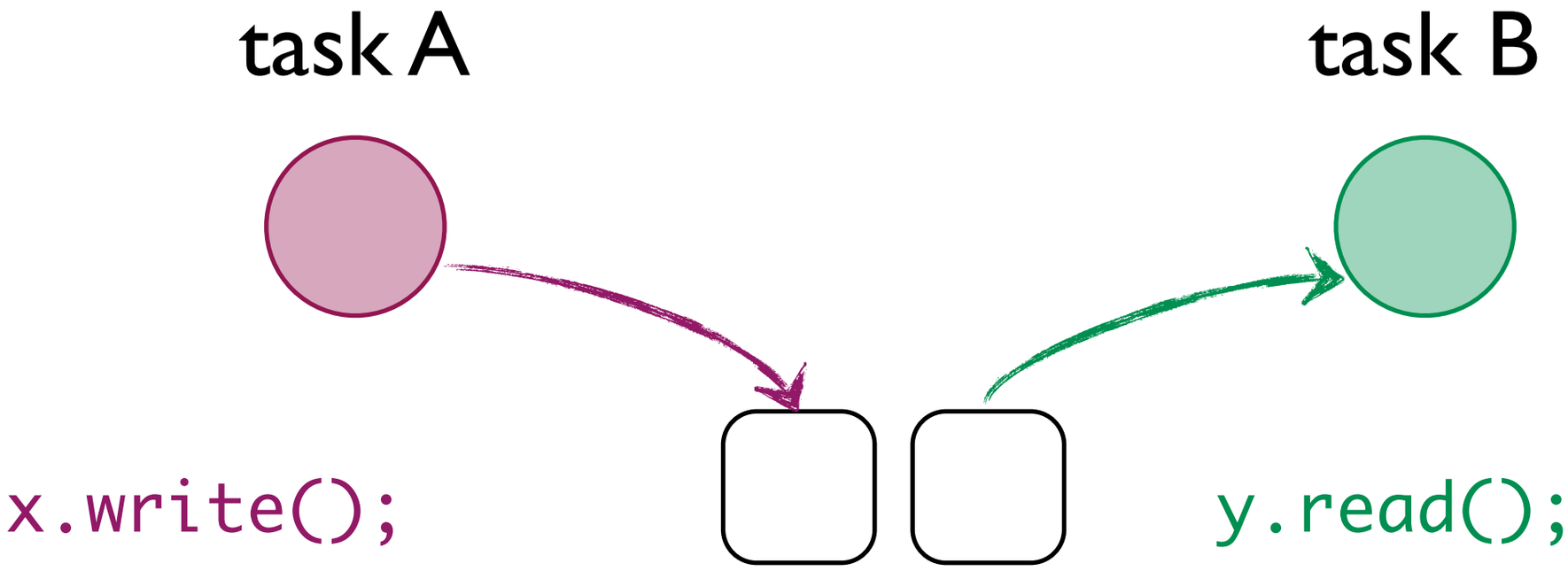
- Can we remove synchronization for x, y?



☑ Different Objects

Example

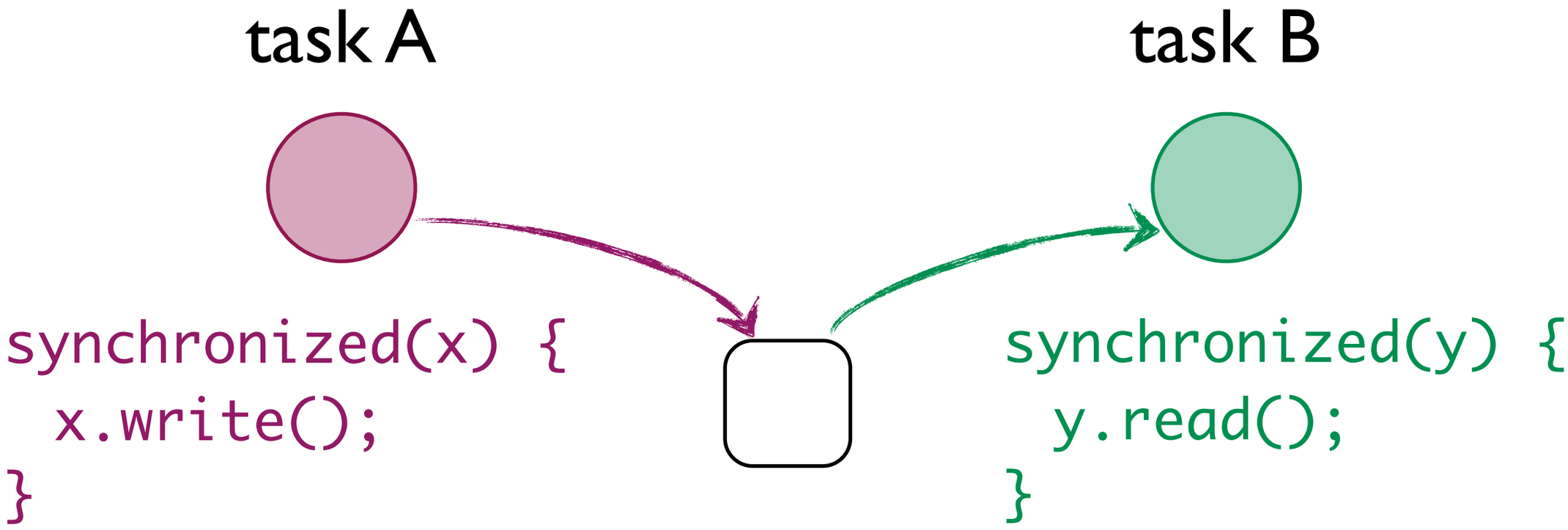
- Can we remove synchronization for x, y?



☑ Different Objects

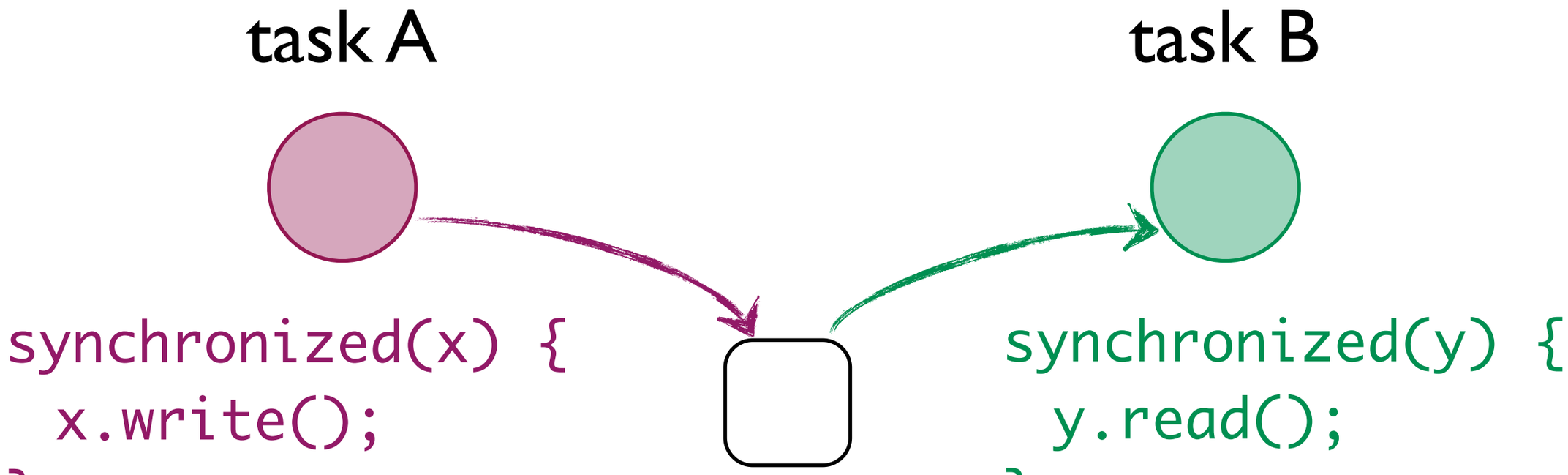
Example

- Can we remove synchronization for x, y?



Example

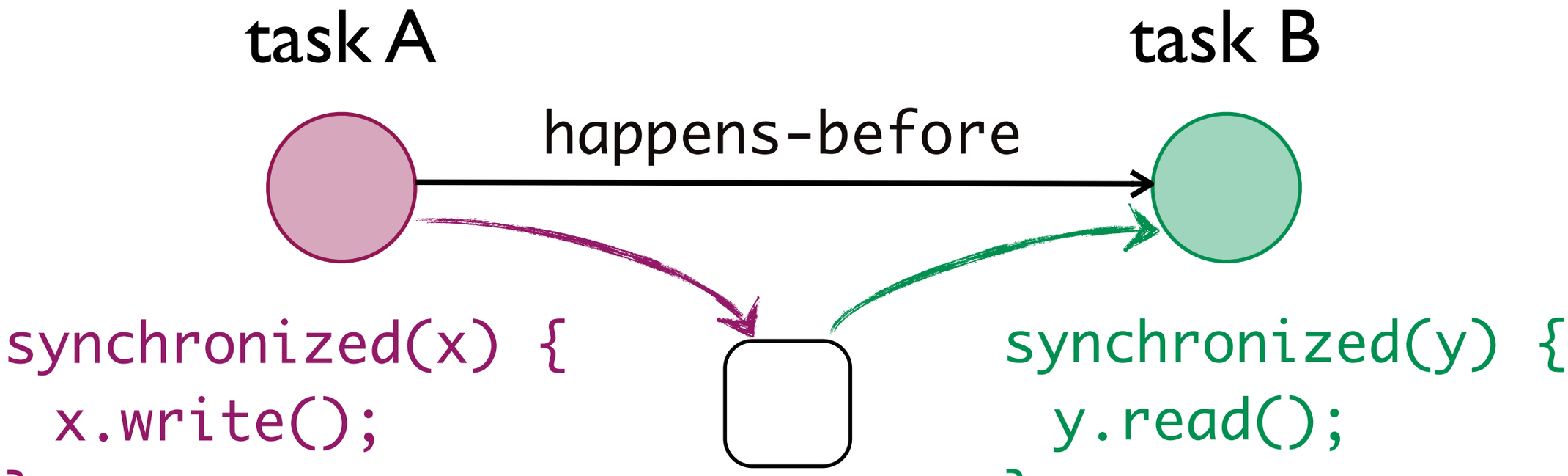
- Can we remove synchronization for x, y?



☒ Aliased

Example

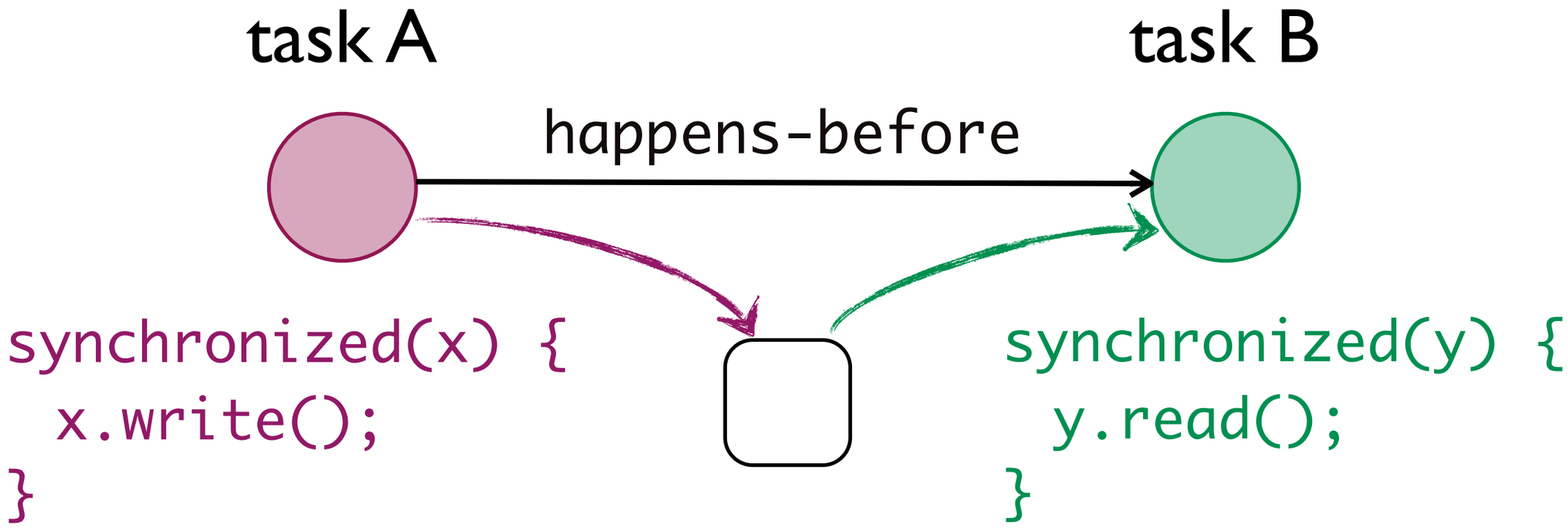
- Can we remove synchronization for x, y?



☒ Aliased

Example

- Can we remove synchronization for x, y?

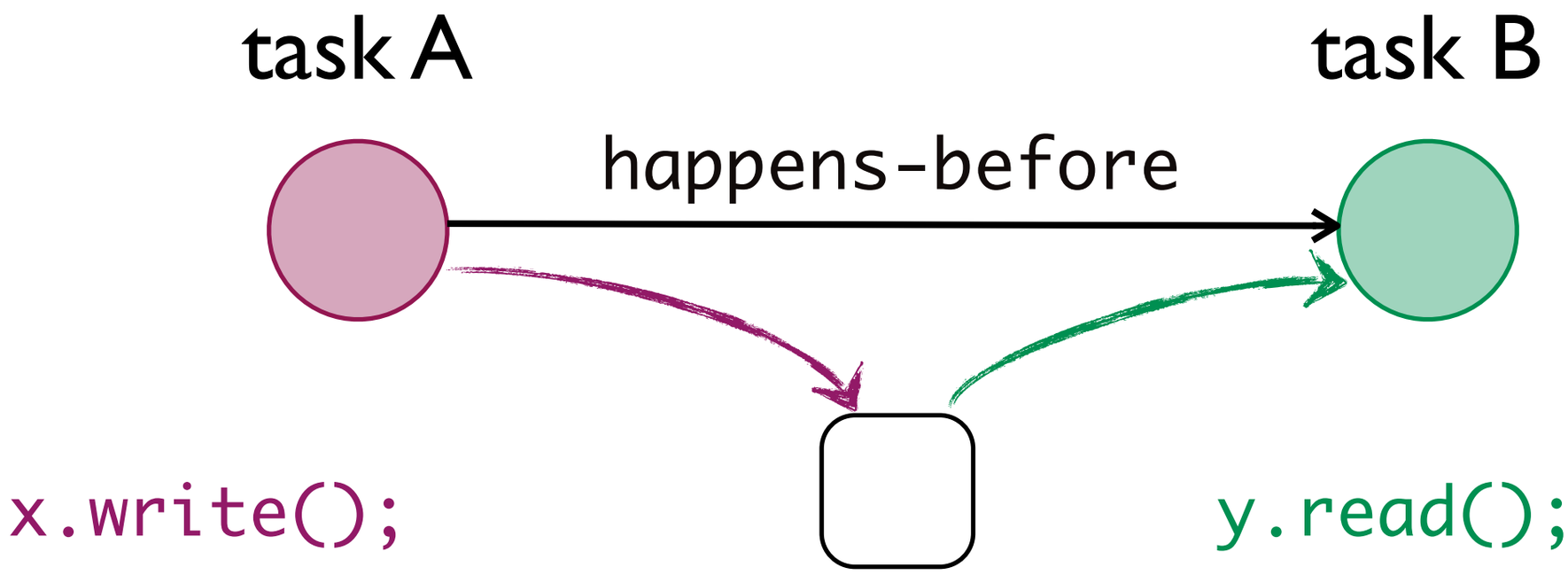


☒ Aliased

☑ Ordered memory access

Example

- Can we remove synchronization for x, y?



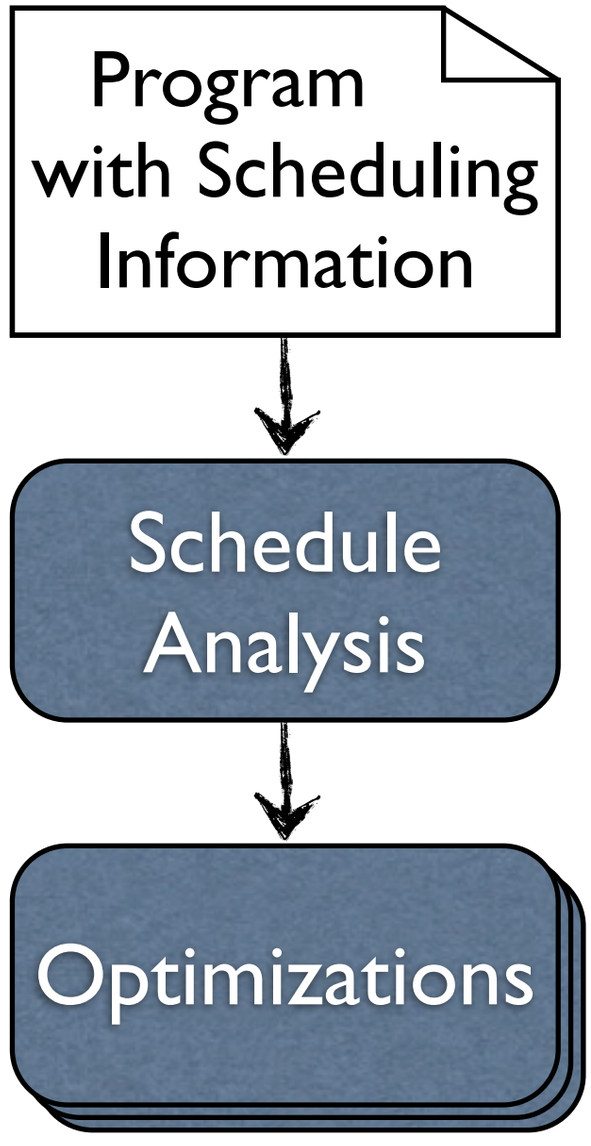
☒ Aliased

☑ Ordered memory access

Motivation

- Optimizations profit from static knowledge about runtime schedules
- Optimizations today must reinvent own analyses
- Our goal: factor out analysis of task schedules
  - Simplification + integration of optimizations
  - Task ordering information increases optimization potential

Schedule Analysis Overview



Outline

- Motivation
- Explicit Scheduling
- Schedule Analysis
- Optimizing Strong Atomicity Overhead
- Related Work
- Concluding Remarks

Explicit Scheduling Model

- A program representation that:
  - Contains explicit scheduling information
  - Allows for static reasoning
- General enough for *structured* (fork/join, Cilk, OpenMP) and *unstructured* parallelism (threads)
- Pre-processing step transforms traditional programs into programs with explicit scheduling

Explicit Scheduling

Explicit Scheduling

- A `task` method is similar to a regular method:
  - code that is executed in the context of `this`

Explicit Scheduling

- A `task` method is similar to a regular method:
  - code that is executed in the context of `this`
- Instead of *calling* a task method, one *schedules* it for later execution:

```
Activation b = schedule obj.bar(42);
```

Explicit Scheduling (2)

Explicit Scheduling (2)

- →-statement creates explicit happens-before relationship:

```
a → b;
```

Explicit Scheduling (2)

- →-statement creates explicit happens-before relationship:

```
a → b;
```

- Implicit happens-before relationship between scheduling task and scheduled task

Explicit Scheduling (2)

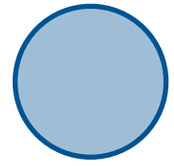
- →-statement creates explicit happens-before relationship:

```
a → b;
```

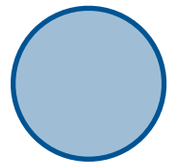
- Implicit happens-before relationship between scheduling task and scheduled task

- At runtime, scheduler constantly chooses executable activations

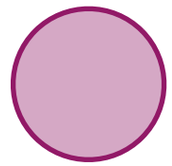
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```




```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



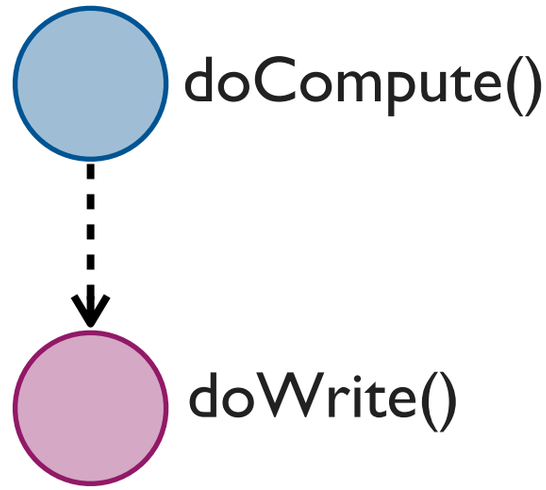
doCompute()



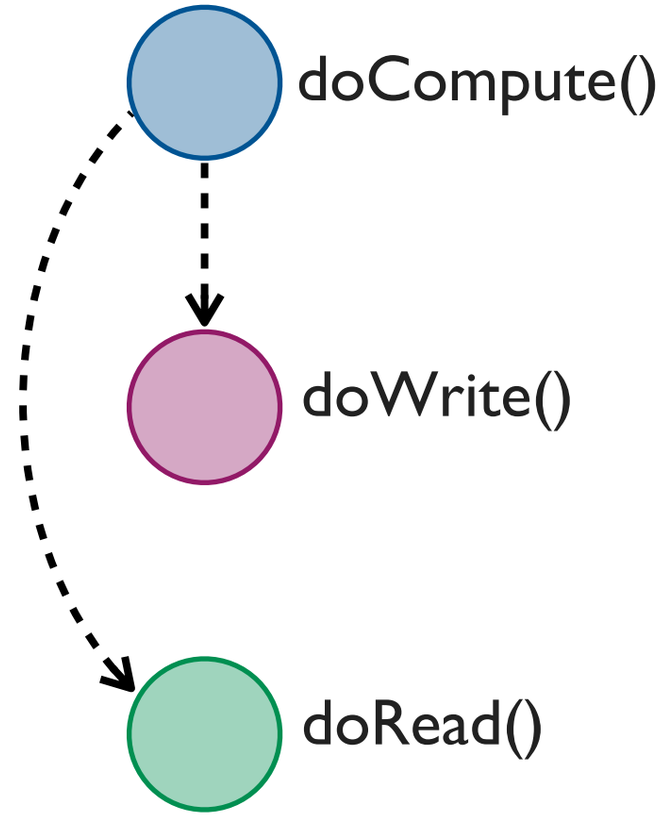
doWrite()

```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```

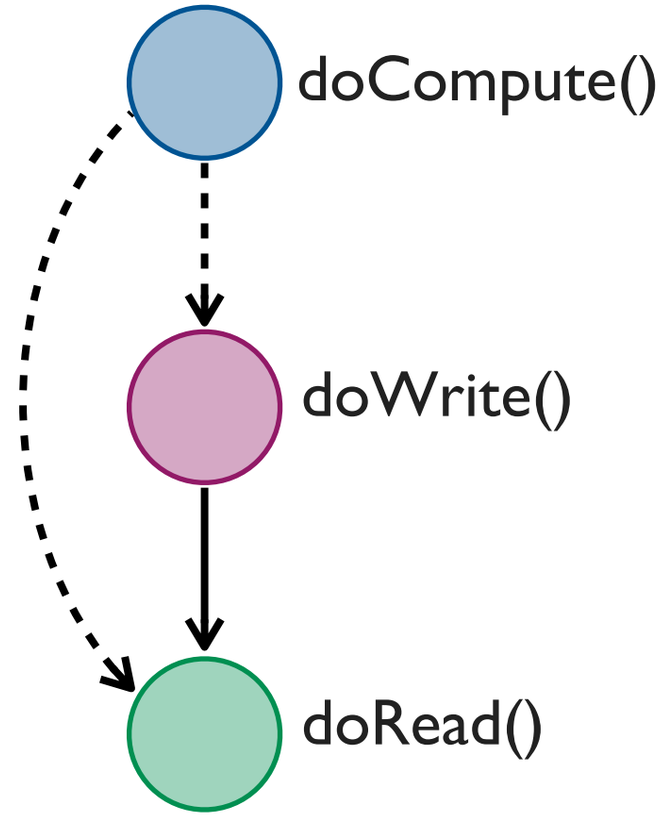
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



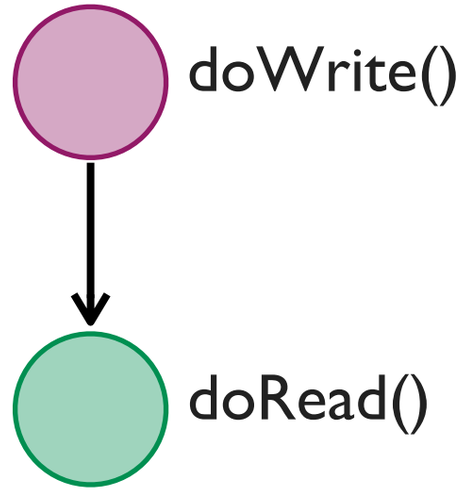
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



Outline

- Motivation
- Explicit Scheduling
- Schedule Analysis
- Optimizing Strong Atomicity Overhead
- Related Work
- Concluding Remarks

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
{ ...
  p1: x.use
  ...
}
```

```
{ ...
  p2: y.use
  ...
}
```

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

```
{ ...
  p1: x.use
  ...
}
```

```
{ ...
  p2: y.use
  ...
}
```

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

☒ Same Object

```
{ ...
  p1: x.use
  ...
}
```

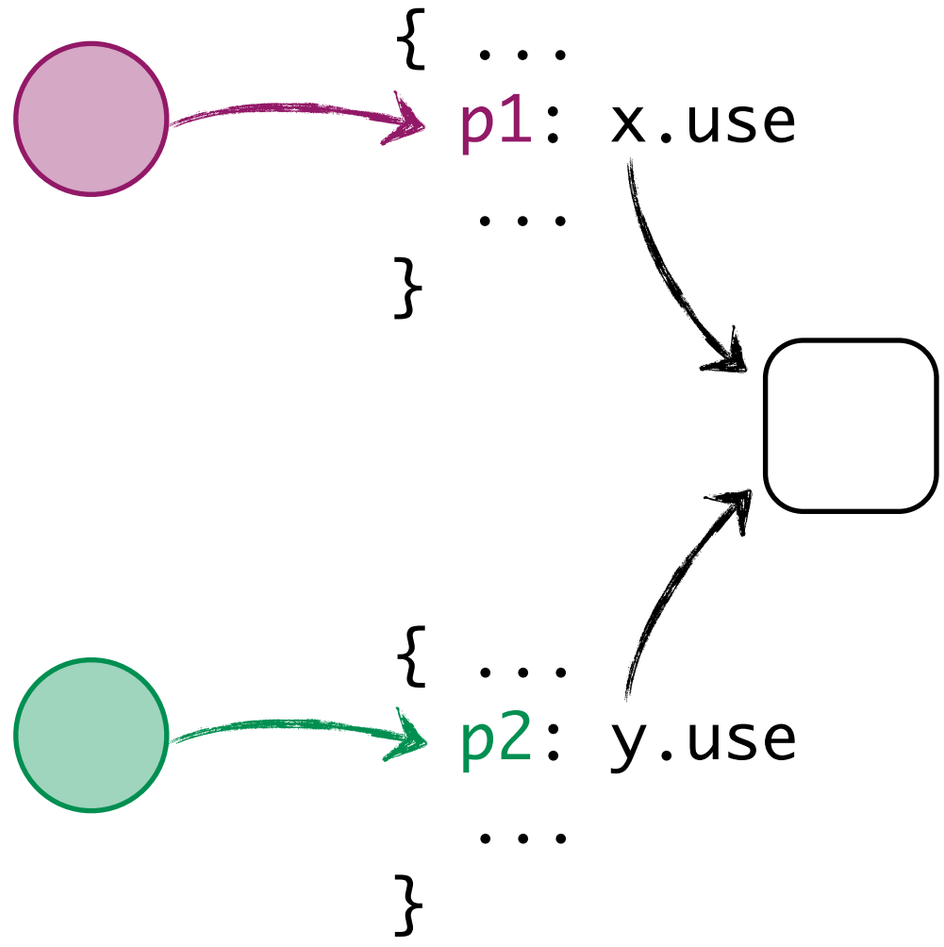
```
{ ...
  p2: y.use
  ...
}
```

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

- ☒ Same Object
- ☒ Different Activations

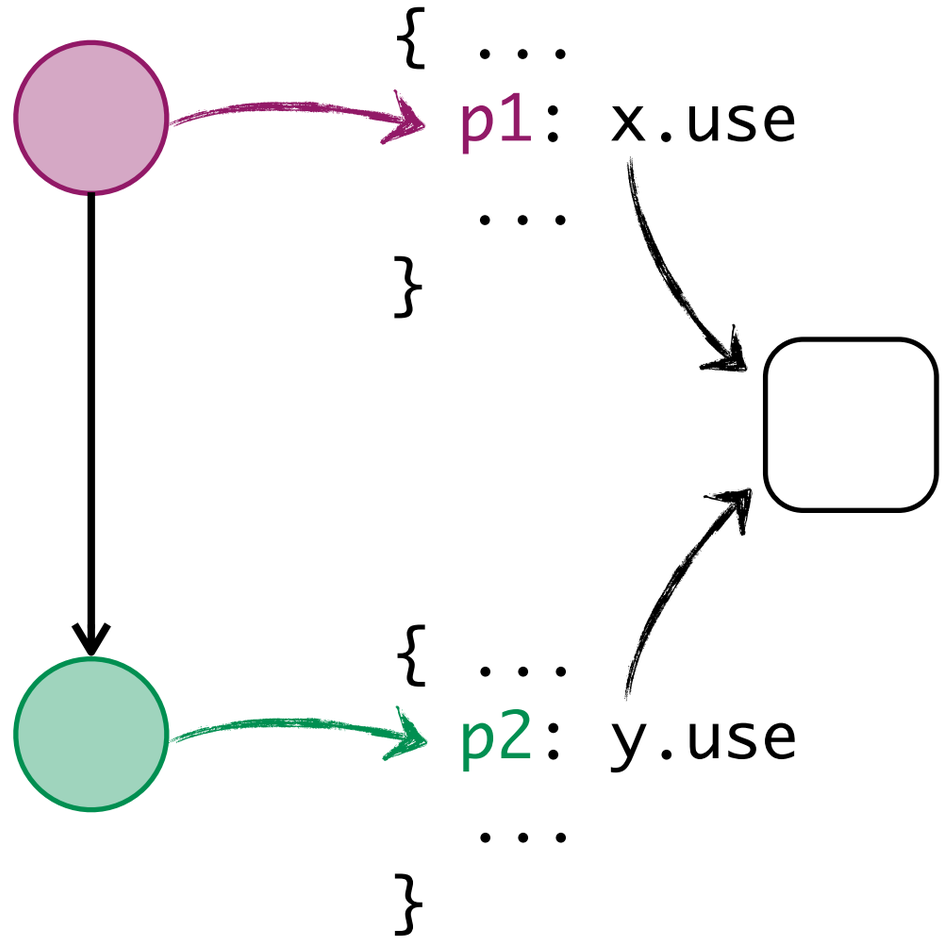


Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

- ☒ Same Object
- ☒ Different Activations

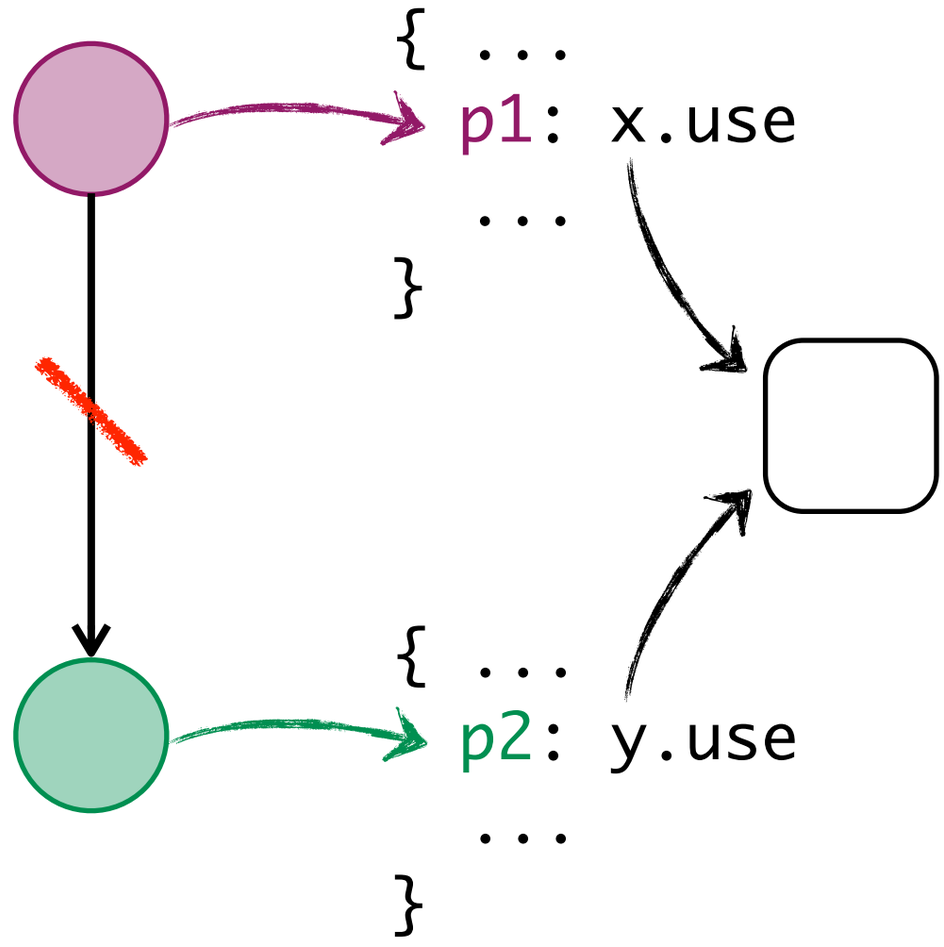


Generic Optimization Question

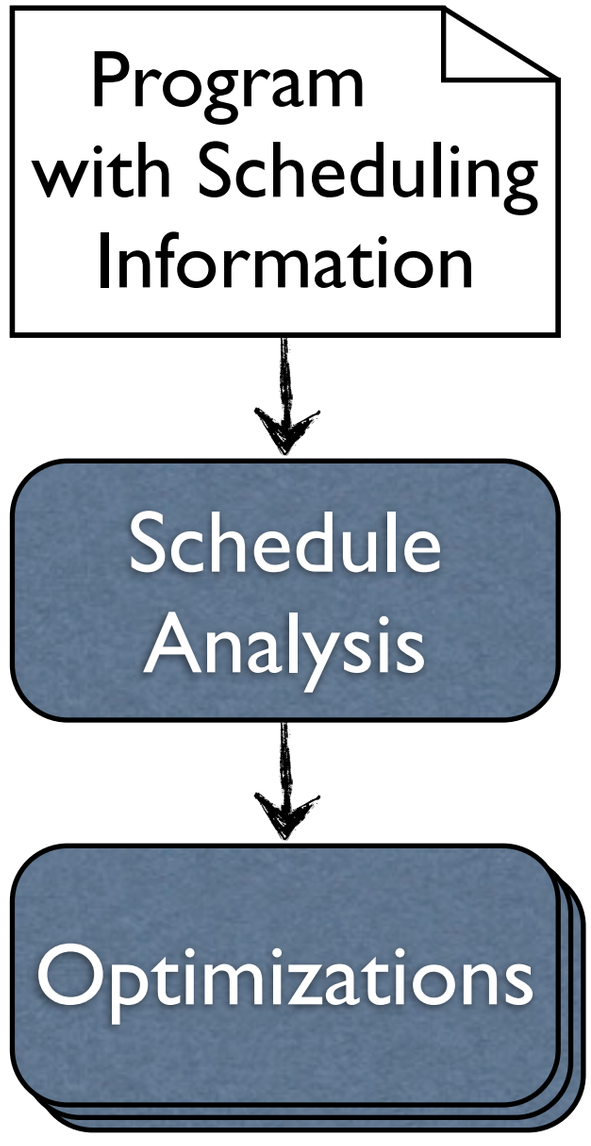
May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

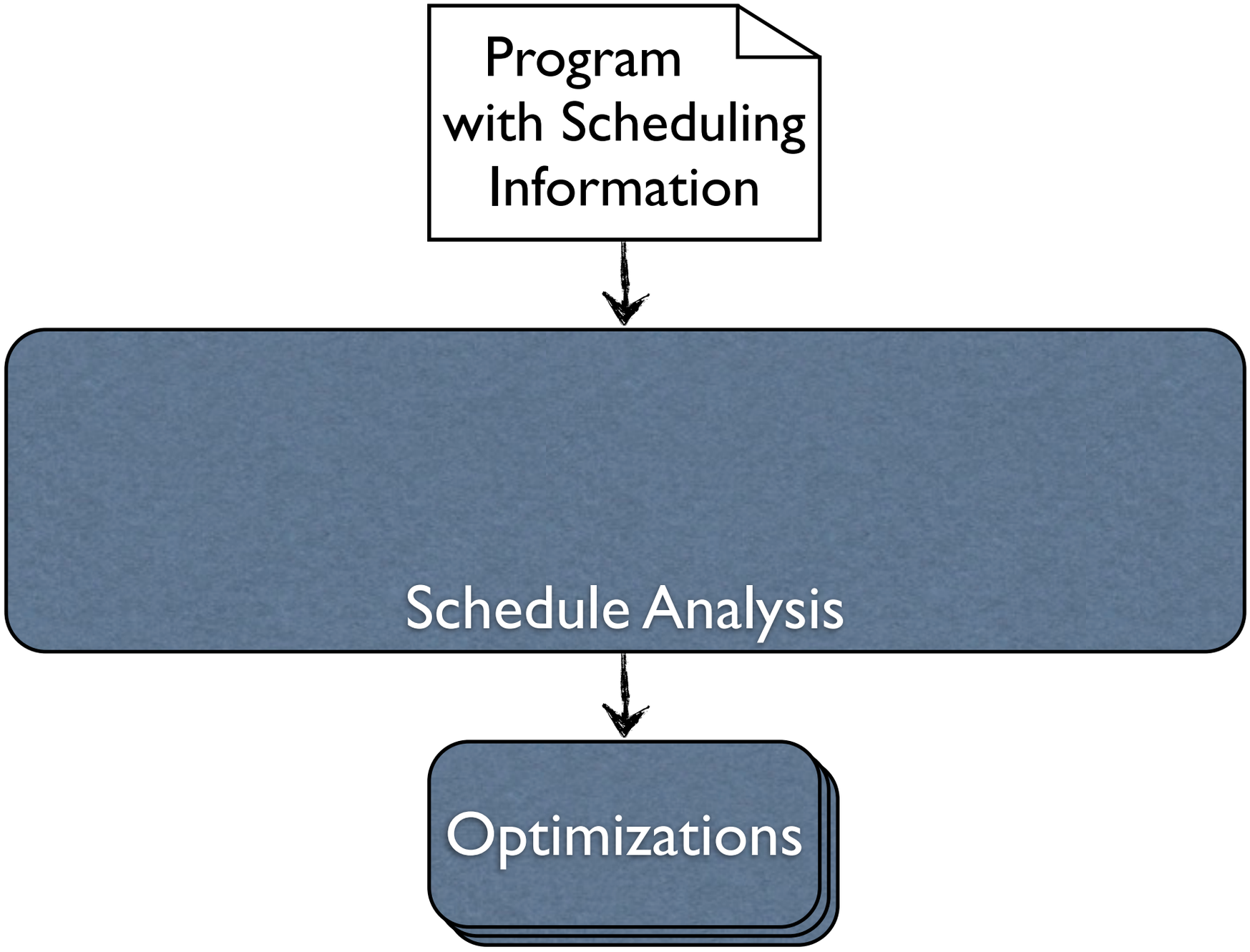
- ☒ Same Object
- ☒ Different Activations
- ☒ Not Ordered



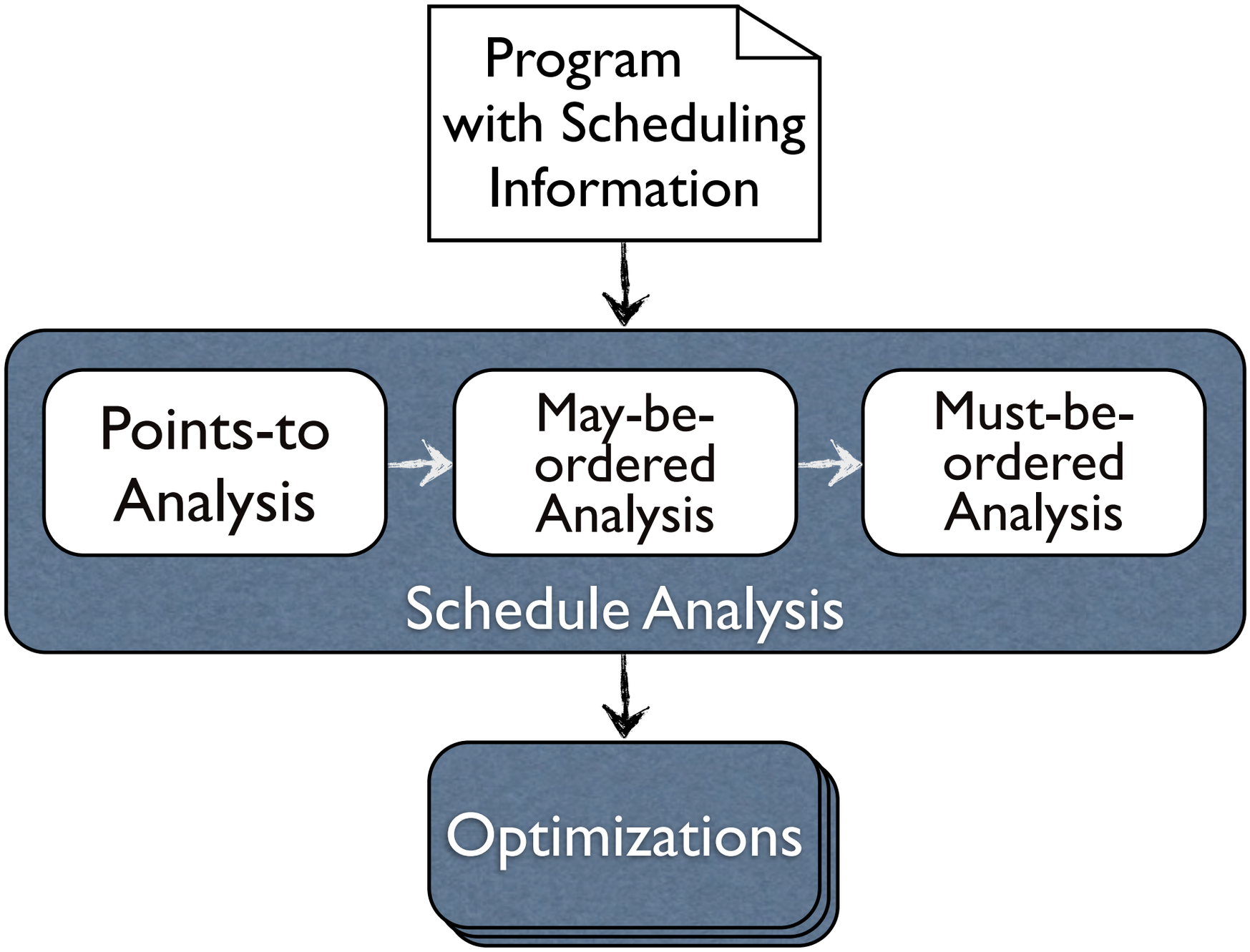
Schedule Analysis Overview



Schedule Analysis Overview



Schedule Analysis Overview



Points-to Analysis

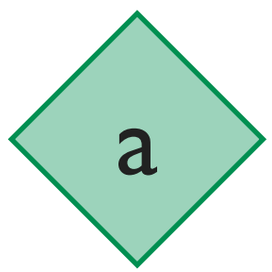
- Computes points-to sets for each program variable
- →-statements handled in next phase
- Treats `schedule` statements as method calls
  - Parameters are bound at schedule-time
  - Flow-insensitive with respect to calls

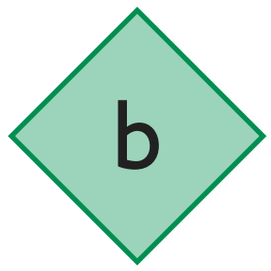
Points-to Analysis

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```

Points-to Analysis

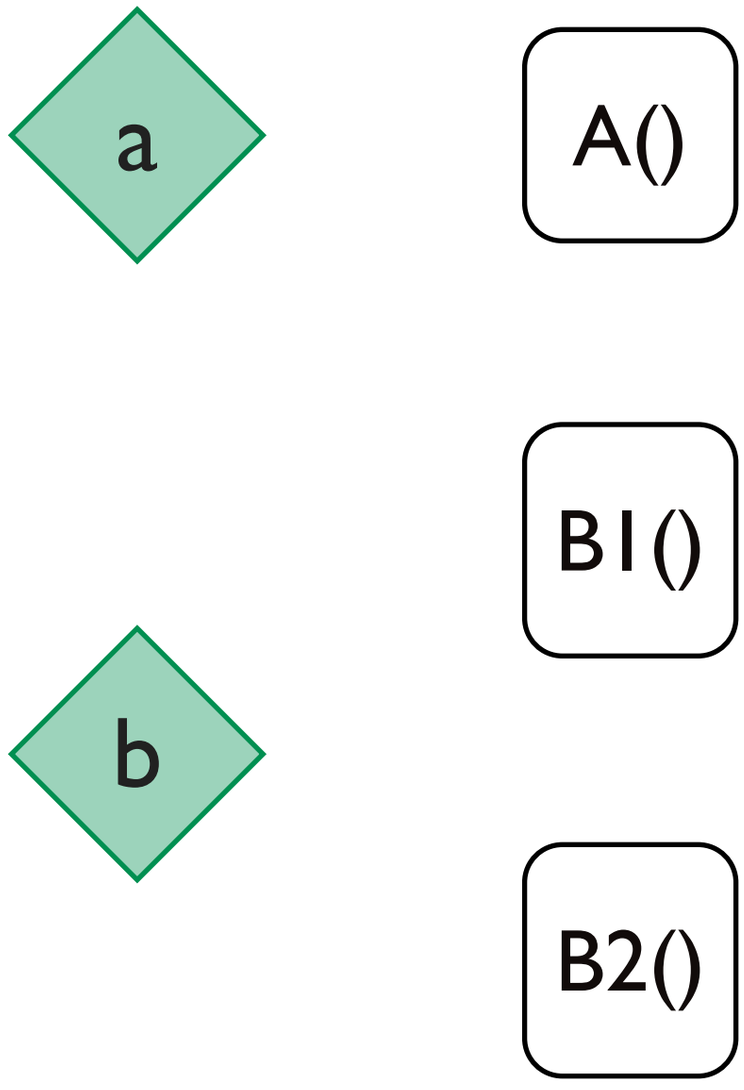
```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```





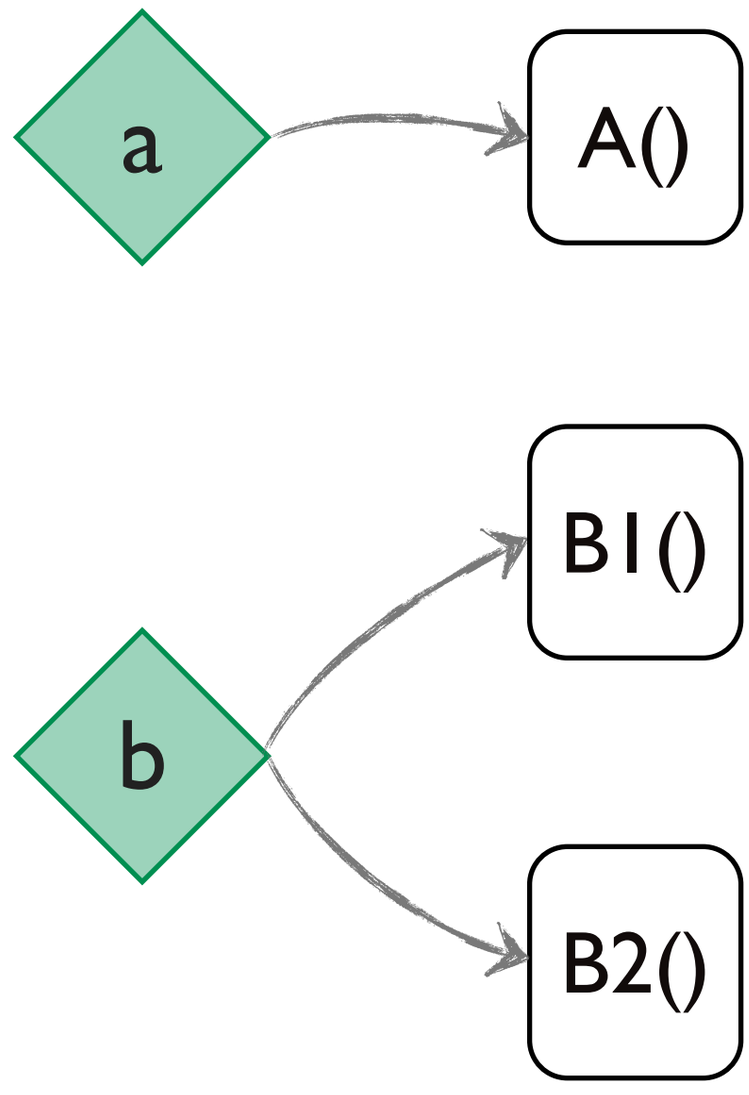
Points-to Analysis

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



Points-to Analysis

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```

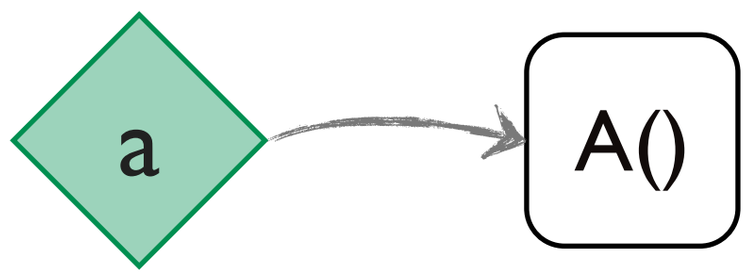


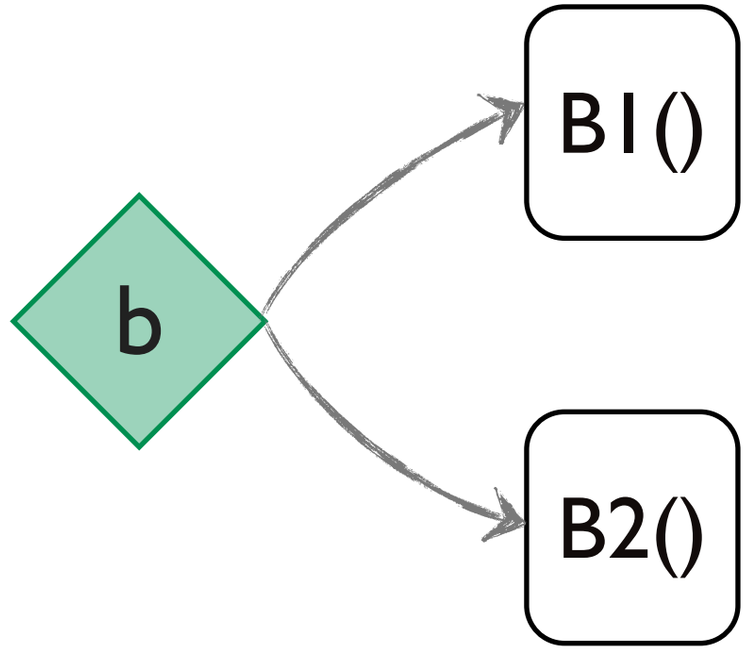
May-be-ordered Analysis

- Analyze happens-before relationships
  - Implicit creation edges
  - Explicit arrow statements
- Compute read/write sets for each activation

May-be-ordered Analysis

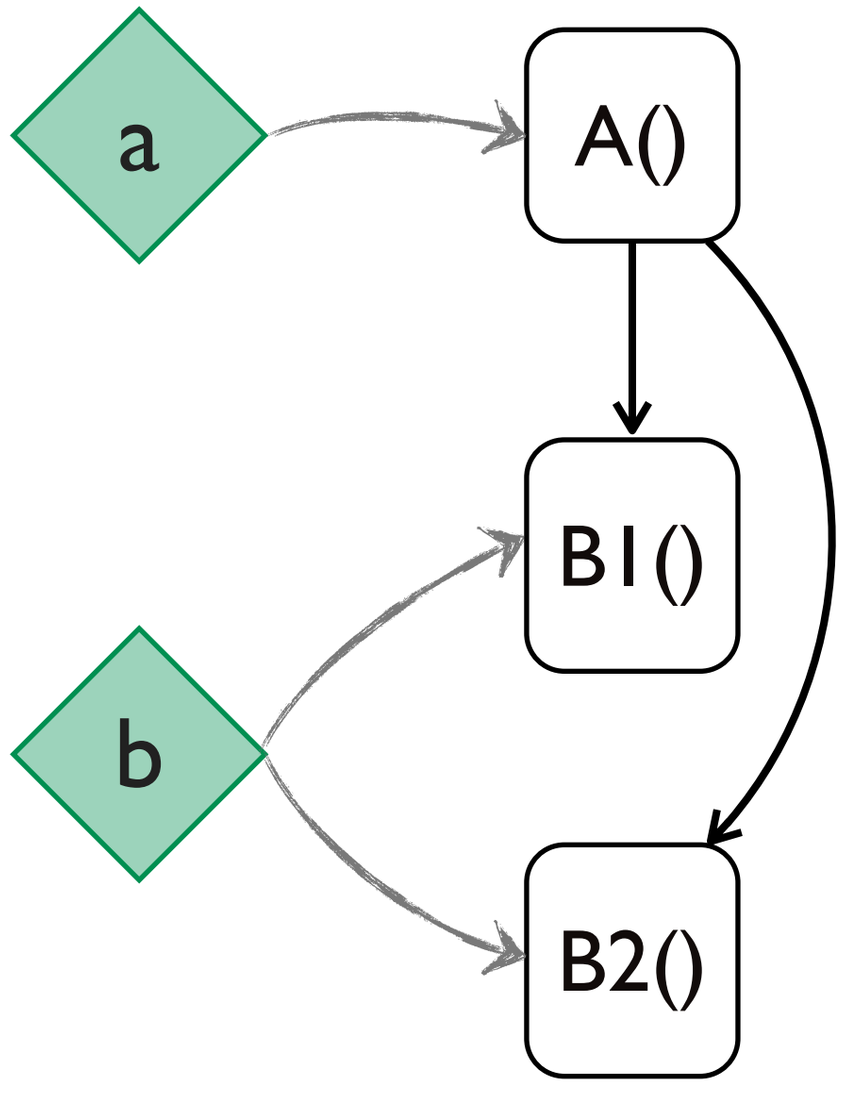
```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```





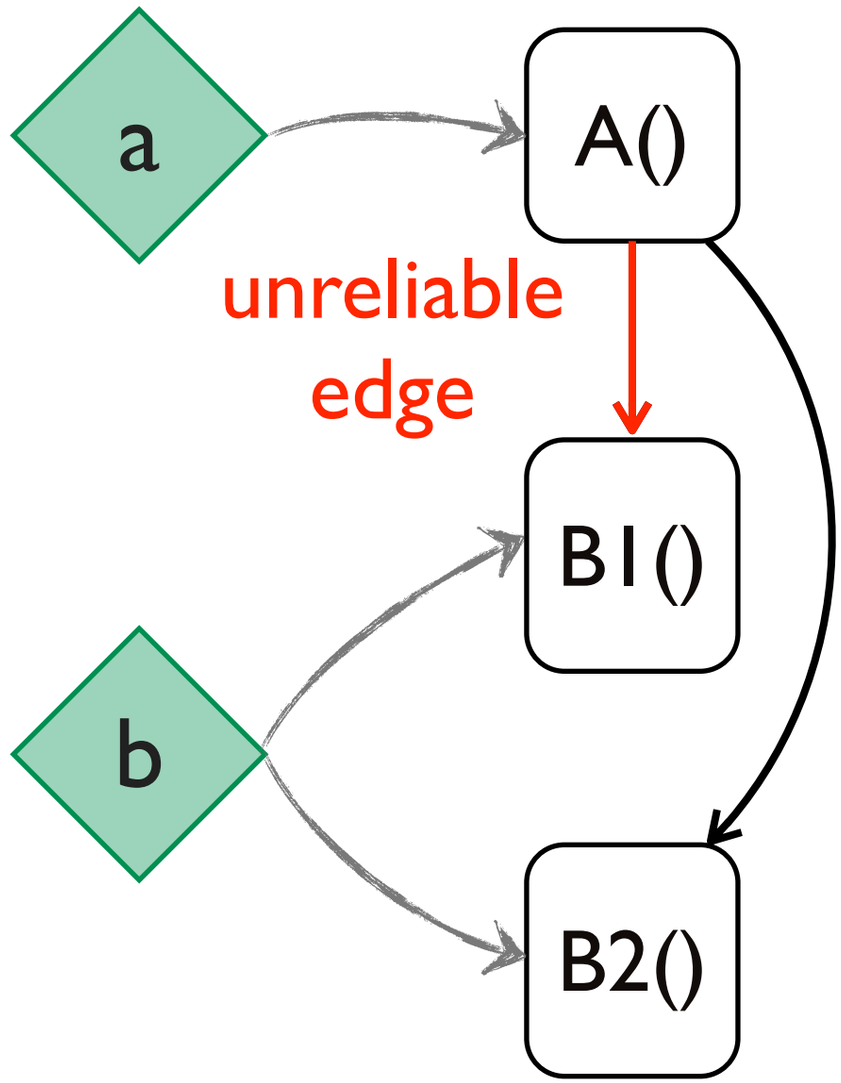
May-be-ordered Analysis

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



May-be-ordered Analysis

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



Conservative Approximation

Conservative Approximation

- May-be-ordered Analysis:
  - over-approximates happens-before edges
  - under-approximates parallelism

Conservative Approximation

- May-be-ordered Analysis:
  - over-approximates happens-before edges
  - under-approximates parallelism

➡ But: want to know if two activations *must* be ordered

Must-be-ordered Analysis

Must-be ⊆ Reality ⊆ May-be

in # of happens-before edges

Question: how to get from may-be ordered to must-be ordered?

Must-be-ordered Analysis

Must-be-ordered Analysis

- *Trivial* **solution: remove *all* edges**

Must-be-ordered Analysis

- *Trivial* **solution: remove *all* edges**
- *Current* **solution: filter out edges**
  - No conditional edges
  - Only edges with single source and target objects
  - Analyze simple loops, but no recursion

Outline

- Motivation
- Explicit Scheduling
- Schedule Analysis
- Optimizing Strong Atomicity Overhead
- Related Work
- Concluding Remarks

Concurrency Control Mechanisms

- Two common concurrency control mechanisms for shared-memory models:
  - *Locks* **with synchronized blocks**
    - ➡ **Synchronization removal optimization**
  - *Software Transactional Memory* **(STM)**
    - ➡ **Reducing strong atomicity overhead**

Strong Atomicity

- In a strong atomic model, every memory access must be treated *as if* it occurred inside a transaction
- e.g., a write `x.f = 3;` outside a transaction is equivalent to:

```
atomic { x.f = 3; }
```

- Without optimization:
  - Overhead on every single read/write outside a transaction

Reducing Strong Atomicity Overhead

- A read/write *outside a* transaction requires a memory barrier if:
  - any transaction may access same object
  - it may access it concurrently
- Otherwise, we can remove the memory barrier

Reducing Strong Atomicity Overhead

- A read/write *outside a* transaction requires a memory barrier if:
  - any transaction may access same object
  - it may access it concurrently
- Otherwise, we can remove the memory barrier

```
requiresReadBarrier(readBC) :-
    readOutsideTransaction(readBC),
    writeInsideTransaction(writeBC),
    mayInterfere(readBC, writeBC).
```

Outline

- Motivation
- Explicit Scheduling
- Schedule Analysis
- Optimizing Strong Atomicity Overhead
- Related Work
- Concluding Remarks

Related Work

- Pointer Analysis for Parallel Programs [Rugina, Rinard '03]
  - Interference information for fork/join parallelism
- Combined with Escape Analysis [Salcianu, Rinard '01], [Nanda, Ramesh '03]
  - Compute Points-to sets, no ordering
- May-happen-in-parallel [Naumovich et al. '99]
  - For X10 (structured parallelism)
- May-happen-before [Barik '05]
  - Happens-before relations in thread creation trees

Concluding Remarks

- Exposing scheduling constraints to compiler enables static analysis of runtime schedules
- →-statements increase optimization potential
- Schedule analysis factors out common aspects of optimizations
  - Integration into single optimizing compiler
  - E.g., locking and STM in same program

Concluding Remarks

Concluding Remarks

- Exposing scheduling constraints to compiler enables static analysis of runtime schedules
  - Independent from parallelism mechanism
  - e.g., threads, fork/join, intervals, ...

Concluding Remarks

- Exposing scheduling constraints to compiler enables static analysis of runtime schedules
  - Independent from parallelism mechanism
  - e.g., threads, fork/join, intervals, ...
- Schedule analysis factors out common aspects of optimizations
  - Simplifies optimization implementations
  - Integration into single optimizing compiler
  - e.g., locking and STM in same program

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
{ ...
  p1: x.use
  ...
}
```

```
{ ...
  p2: x.use
  ...
}
```

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
```

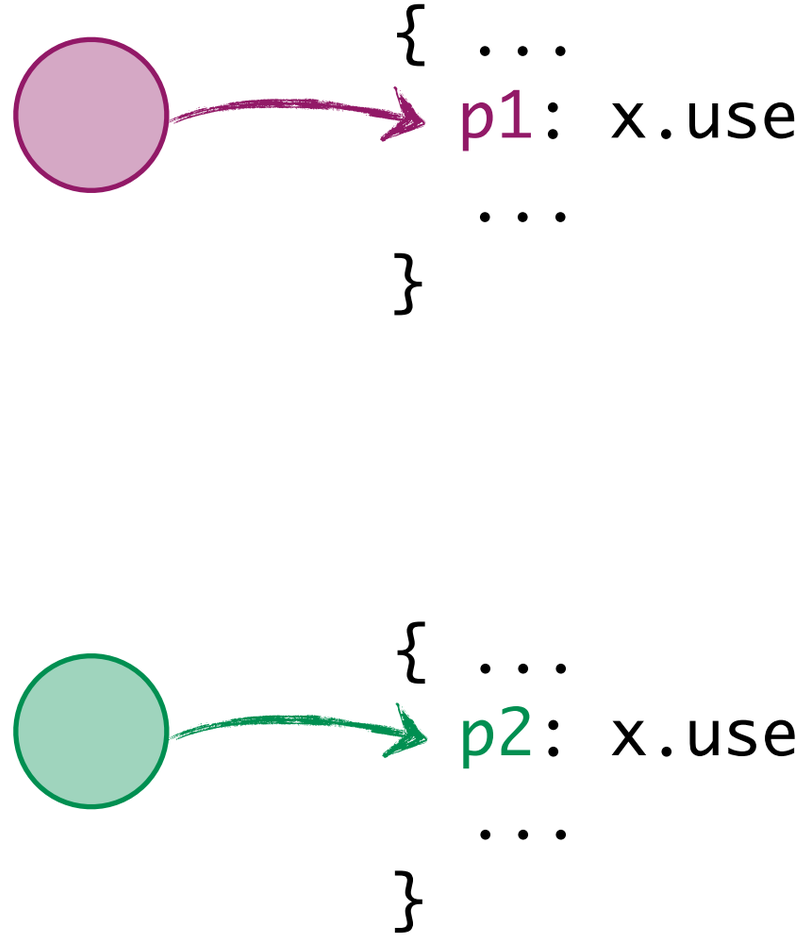
```
{ ...
  p1: x.use
  ...
}
```

```
{ ...
  p2: x.use
  ...
}
```

Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

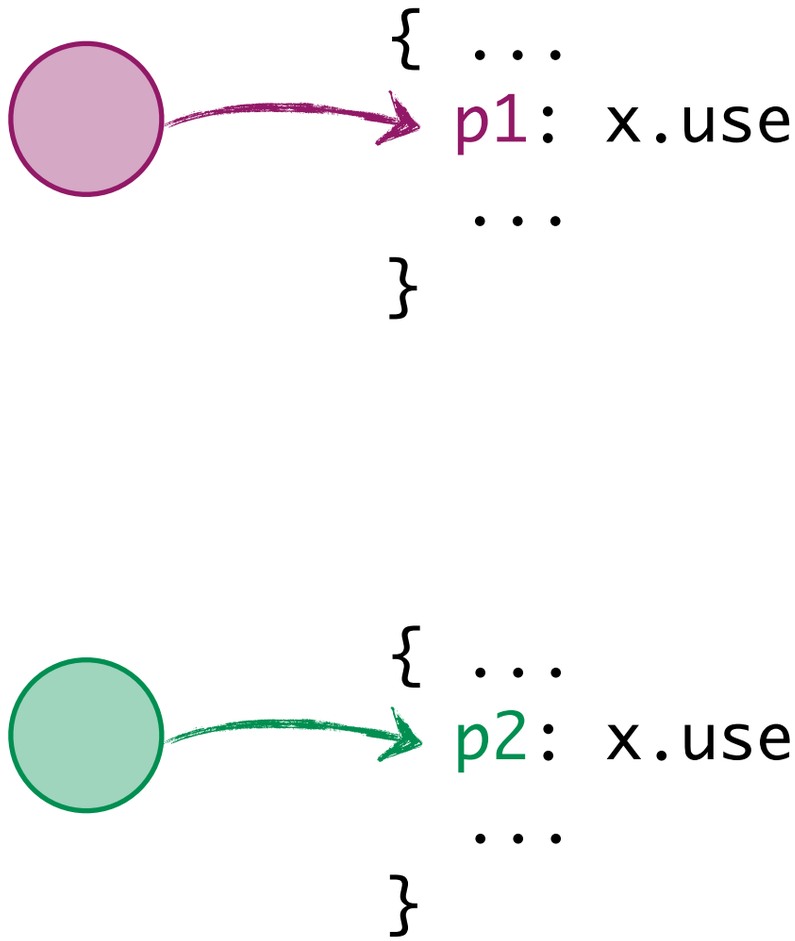
```
mayInterfere(p1, p2) :-
```



Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

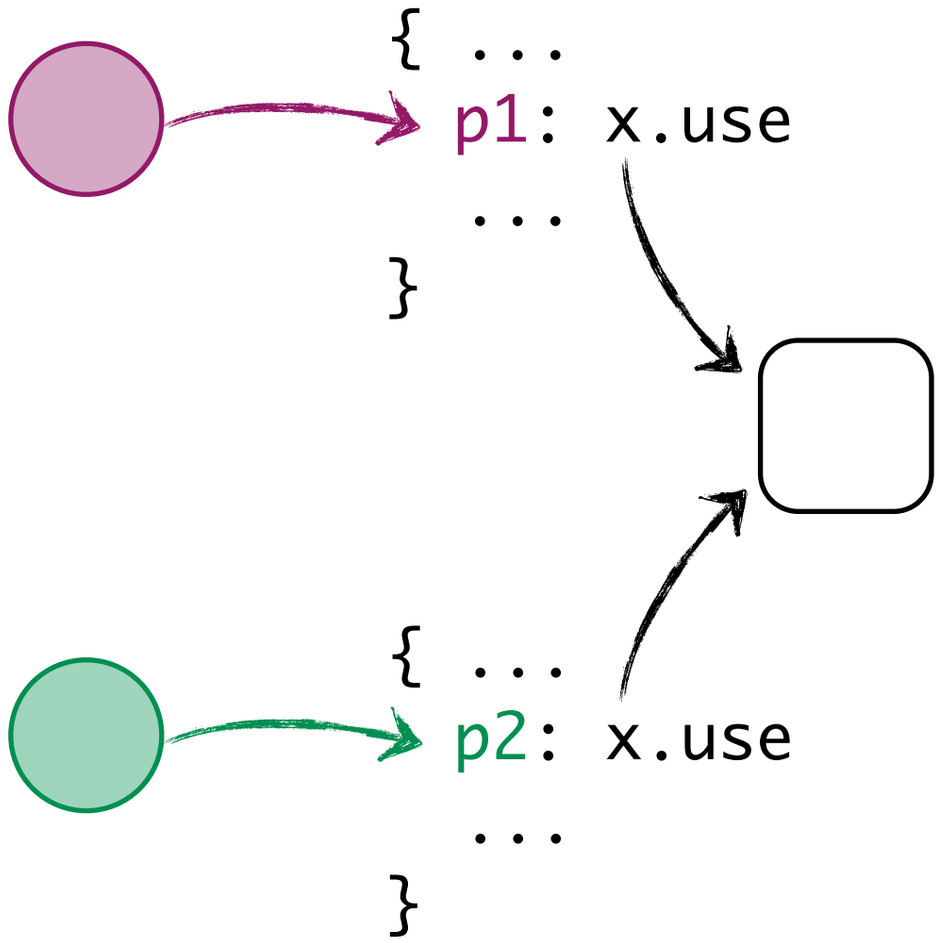
```
mayInterfere(p1, p2) :-
    execute(act1, p1),
    execute(act2, p2),
```



Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

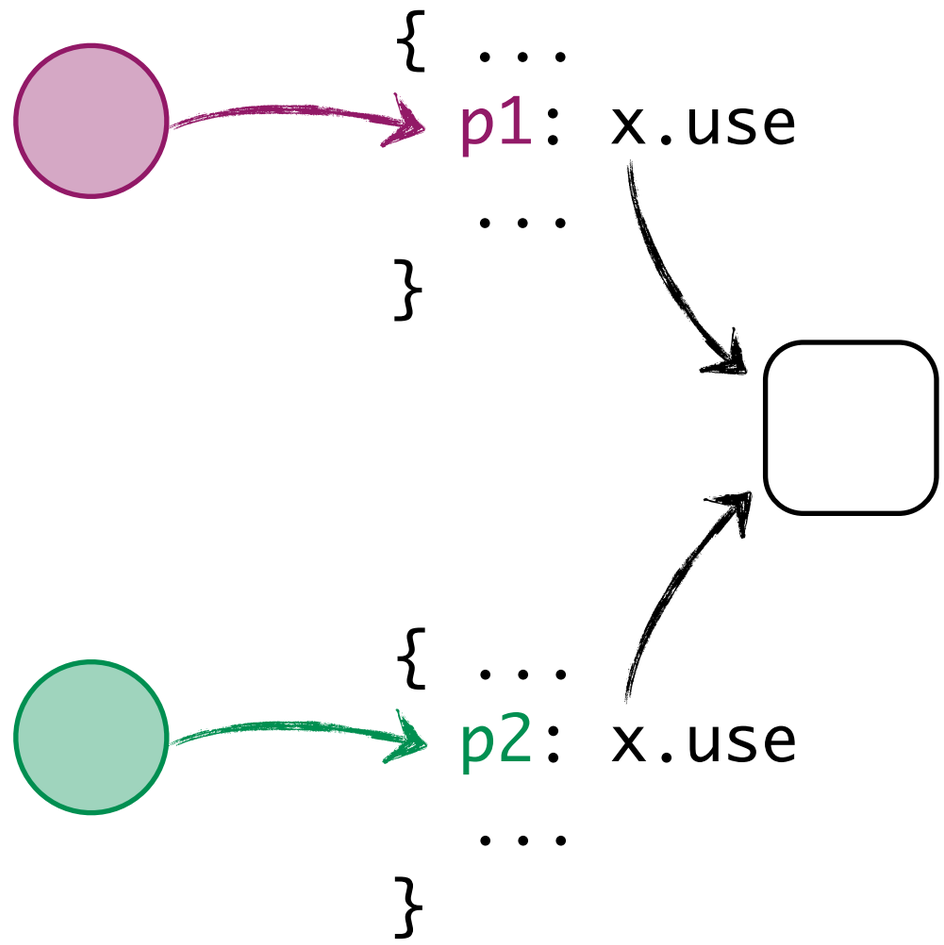
```
mayInterfere(p1, p2) :-
    execute(act1, p1),
    execute(act2, p2),
```



Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

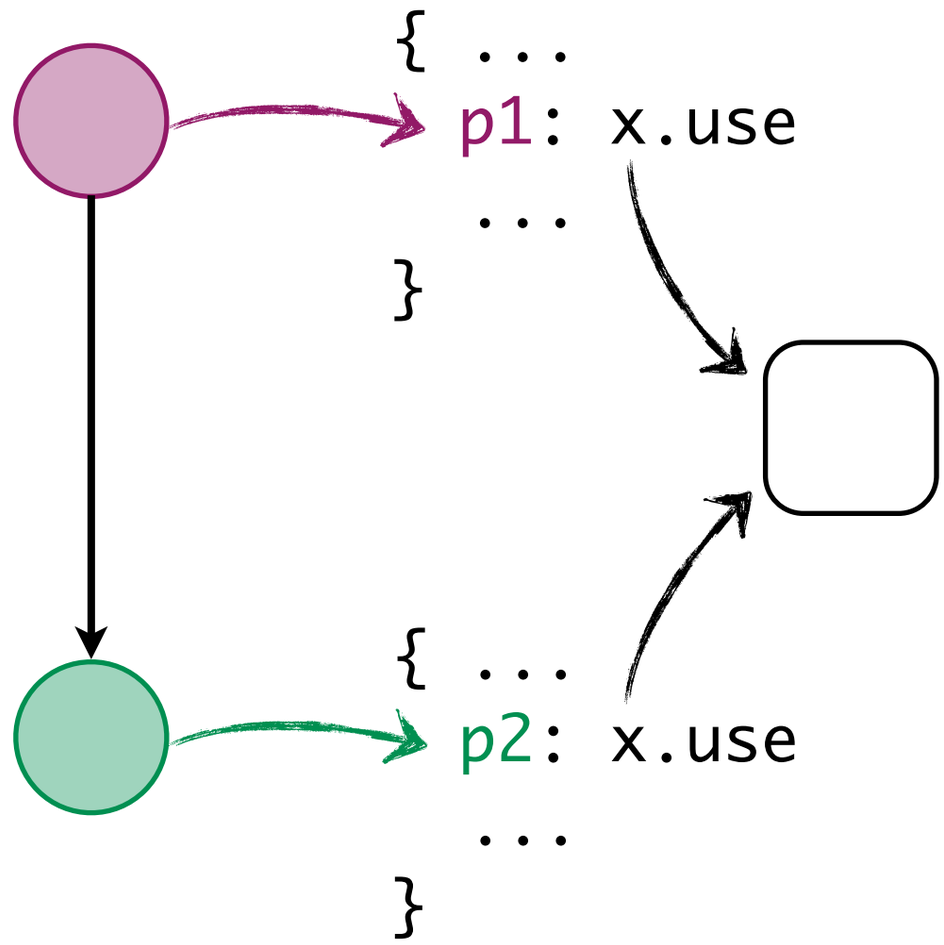
```
mayInterfere(p1, p2) :-
    execute(act1, p1),
    execute(act2, p2),
    access(p1, obj),
    access(p2, obj),
```



Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

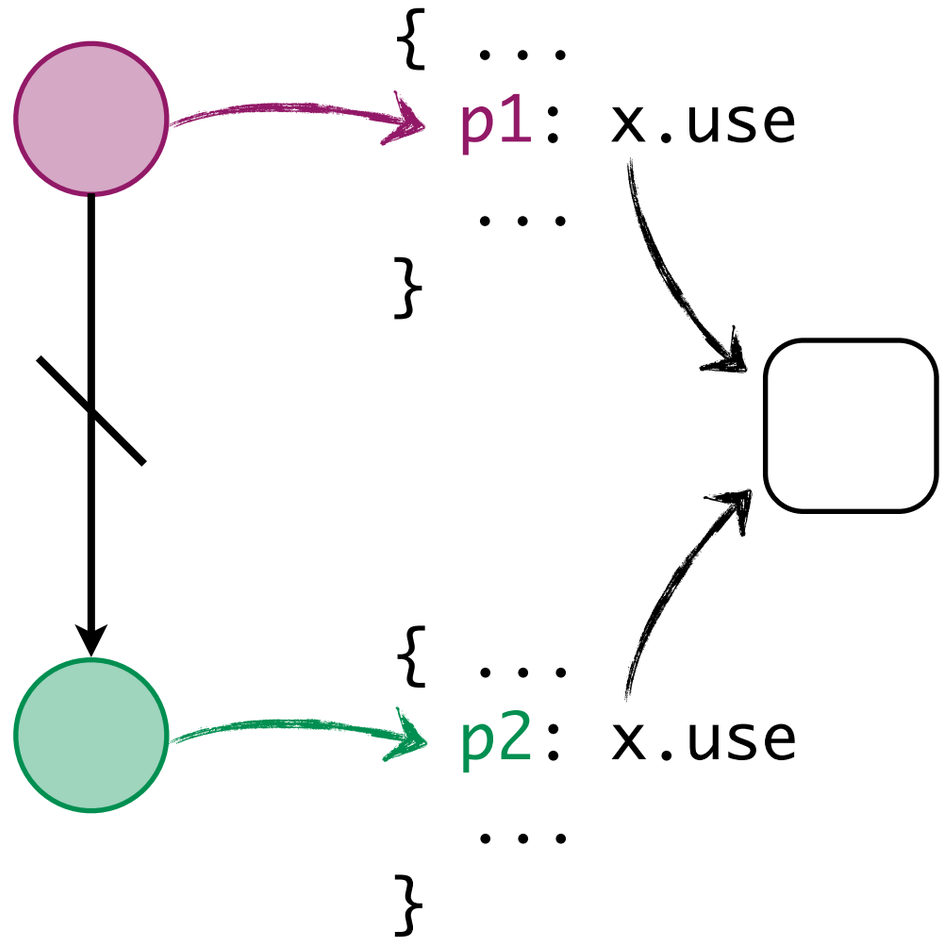
```
mayInterfere(p1, p2) :-
    execute(act1, p1),
    execute(act2, p2),
    access(p1, obj),
    access(p2, obj),
```



Generic Optimization Question

May a memory access at program point p1 interfere with a memory access at program point p2?

```
mayInterfere(p1, p2) :-
    execute(act1, p1),
    execute(act2, p2),
    access(p1, obj),
    access(p2, obj),
    !ordered(act1, act2).
```

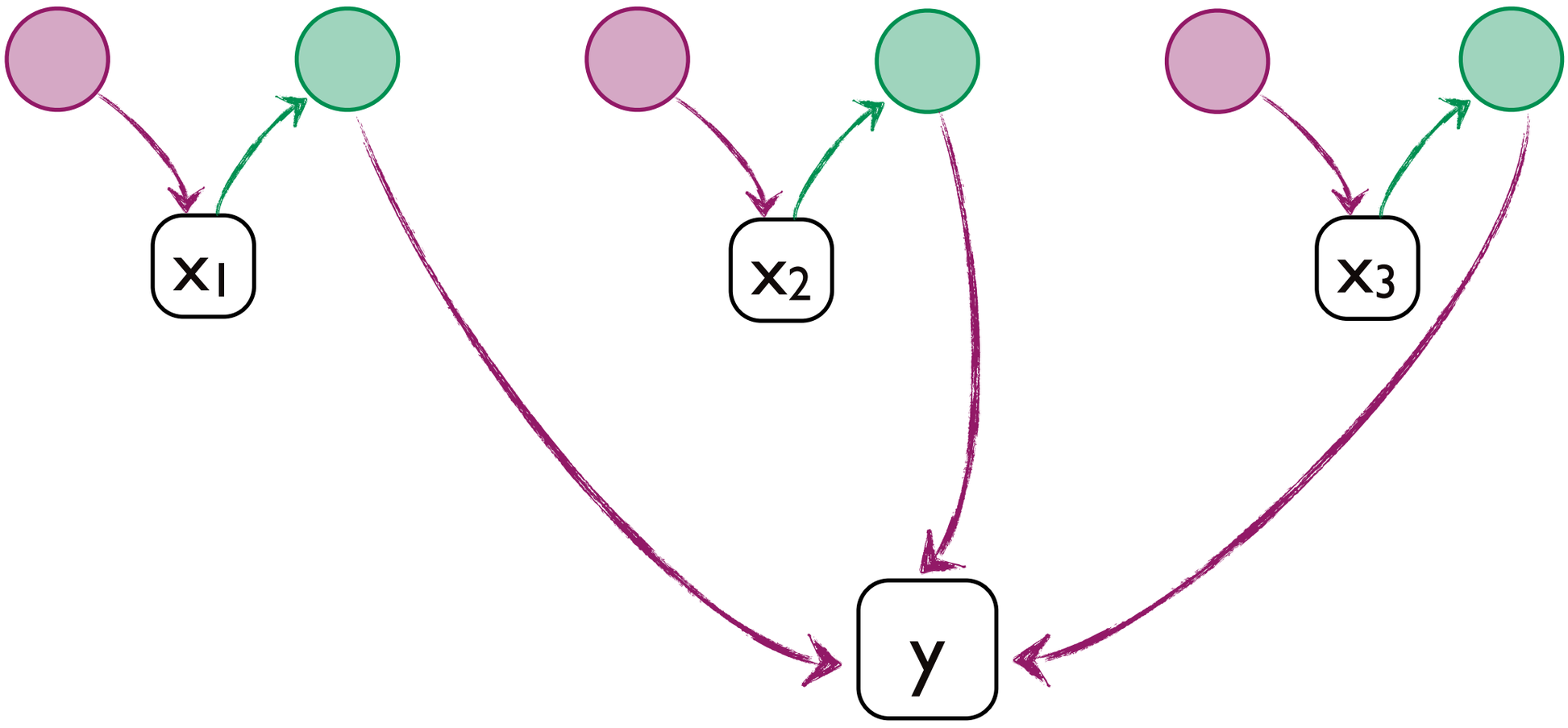


Next Steps

- Identify and analyze special cases/patterns. E.g.:
  - relative ordering of program phases
  - primitive recursion
  - nested loops
- Performance evaluation

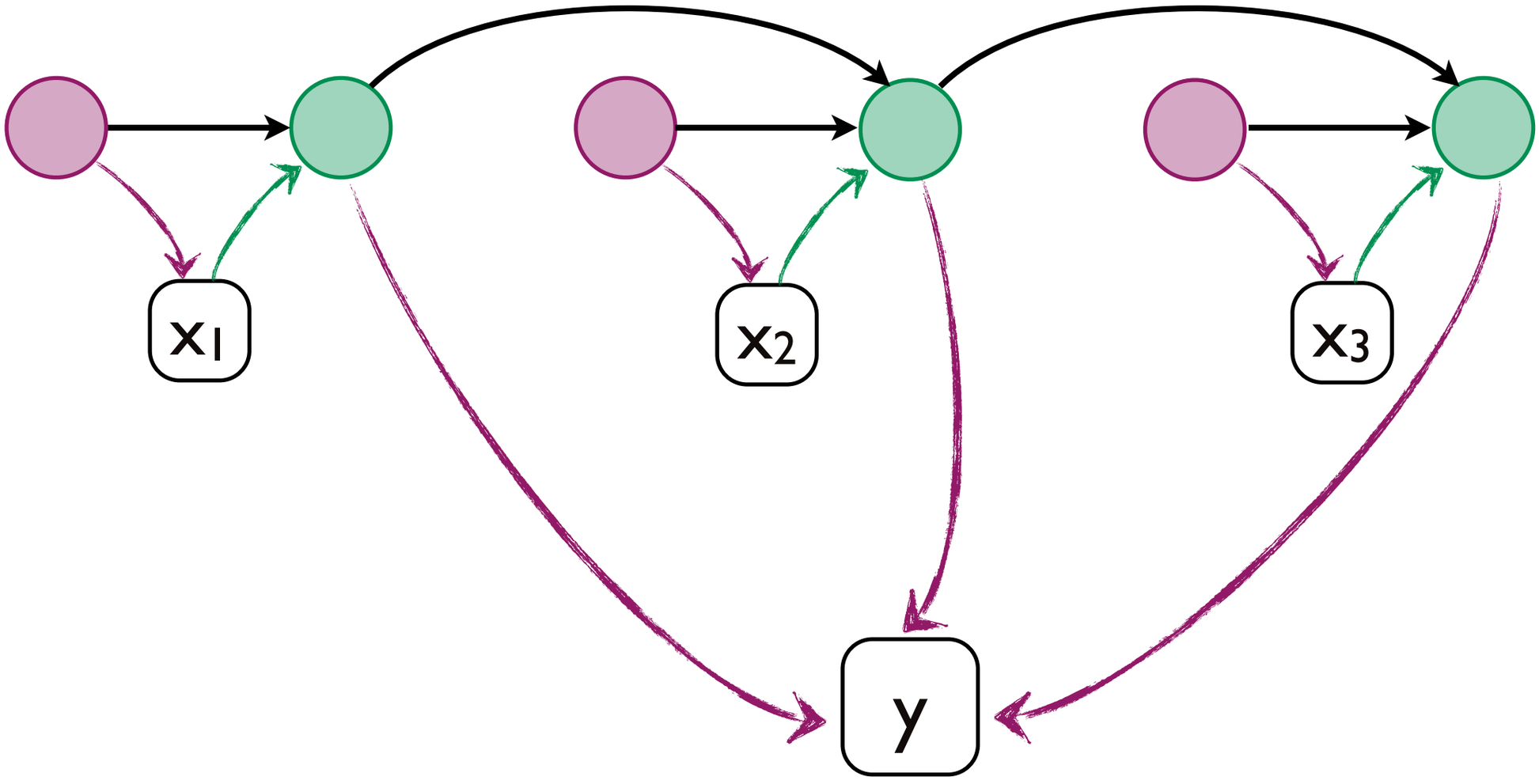
Example

- Do we need to synchronize?



Example

- Do we need to synchronize?



A Note on Notation

- Datalog: formal and executable specification language
- Efficient implementations scale to large real-world program analyses (Whaley&Lam, bddbddb)

A Note on Notation

- Datalog: formal and executable specification language
- Efficient implementations scale to large real-world program analyses (Whaley&Lam, bddbddb)
- Example:

```
trans(x, y)  :- edge(x, y).

trans(x, y2) :- trans(x, y1), edge(y1, y2).
```

A Note on Notation

- Datalog: formal and executable specification language
- Efficient implementations scale to large real-world program analyses (Whaley&Lam, bddbddb)
- Example:

```
trans(x, y)  :- edge(x, y).

trans(x, y2) :- trans(x, y1), edge(y1, y2).
```

- Maps naturally to inference rules:

$$\frac{\texttt{edge}(x, y)}{\texttt{trans}(x, y)} \qquad \frac{\texttt{trans}(x, y1)\ \ \texttt{edge}(y1, y2)}{\texttt{trans}(x, y2)}$$

Concluding Remarks

Concluding Remarks

- Schedule analysis handles different parallelism styles:
  - Threads, fork/join, intervals, ...
  - ... can be intermixed in the same program

Concluding Remarks

- Schedule analysis handles different parallelism styles:
  - Threads, fork/join, intervals, ...
  - ... can be intermixed in the same program
- Synergistic effects between optimizations:
  - Simpler implementations, integrated
  - e.g., locking and STM in the same program

Concluding Remarks

- Schedule analysis handles different parallelism styles:
  - Threads, fork/join, intervals, ...
  - ... can be intermixed in the same program
- Synergistic effects between optimizations:
  - Simpler implementations, integrated
  - e.g., locking and STM in the same program
- Optimizations directly profit from improvements of the schedule analysis

Synchronization Removal

- A monitor enter bytecode is required if parallel activations may try to lock the same object
- Otherwise, it can be removed

Synchronization Removal

- A monitor enter bytecode is required if parallel activations may try to lock the same object
- Otherwise, it can be removed

```
requiredMonitorEnter(enterBC1) :-
   isMonitorEnter(enterBC1),
   isMonitorEnter(enterBC2),
   mayInterfere(enterBC1, enterBC2).
```

Pre-processing

- Compute domains and extract static program facts
- Examples:

Pre-processing

- Compute domains and extract static program facts
- Examples:

```
store(bc:BC, lhs:Var, f:Field, rhs:Var)

new   (lhs:Variable, o:Obj)
```

Pre-processing

- Compute domains and extract static program facts
- Examples:

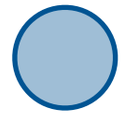
```
store(bc:BC, lhs:Var, f:Field, rhs:Var)

new  (lhs:Variable, o:Obj)
```

```
arrow(lhs:Var, rhs:Var)

schedule(lhs:Var, s:Sig, act:Obj)
```

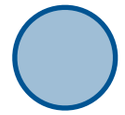
May-be-ordered Analysis

- Compute read/write sets

```
reads(act:Obj, obj:Obj) :-
    execute(act, bc),
    load(bc, v, _, _),
    variablePT(v, obj).
```

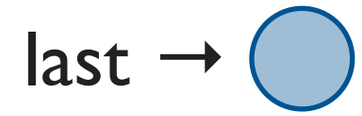


```
class MyClass {

  ...

  task doCompute(Vector in) {

    Activation last = now;

    for(Object o : in) {

      Activation map = sched this.doMap(o);

      Activation write = sched this.doWrite(map);

      map → write;

      last → write;

      last = write;

    }

}}
```



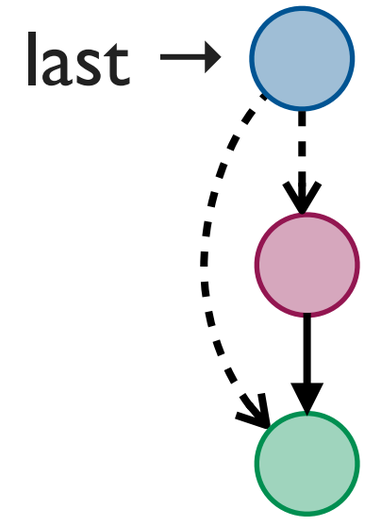
```
class MyClass {

  ...

  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
    }
  }}
```


```
class MyClass {

  ...

  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
    }
}}
```

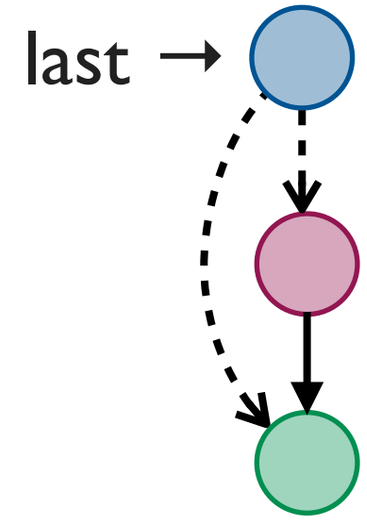



```
class MyClass {

  ...

  task doCompute(Vector in) {

    Activation last = now;

    for(Object o : in) {

      Activation map = sched this.doMap(o);

      Activation write = sched this.doWrite(map);

      map → write;

      last → write;

      last = write;

    }

}}
```

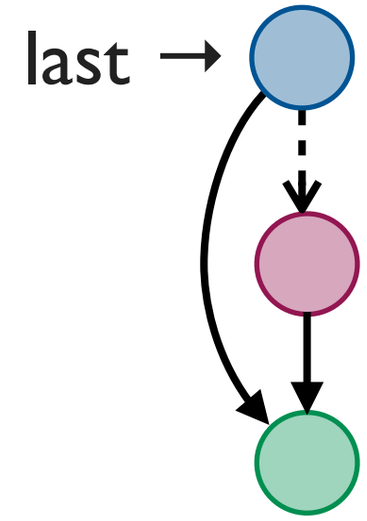
```
class MyClass {

  ...

  task doCompute(Vector in) {

    Activation last = now;

    for(Object o : in) {

      Activation map = sched this.doMap(o);

      Activation write = sched this.doWrite(map);

      map → write;

      last → write;

      last = write;

    }

}}
```



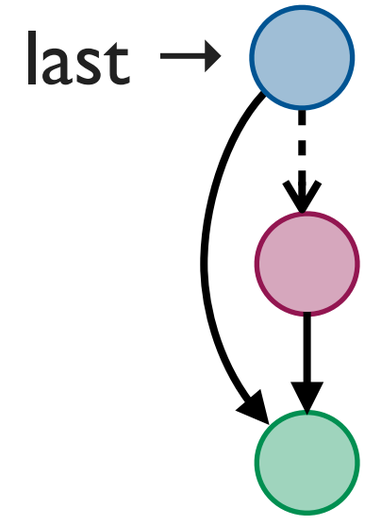
```
class MyClass {
  ...
  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
  }
}}
```



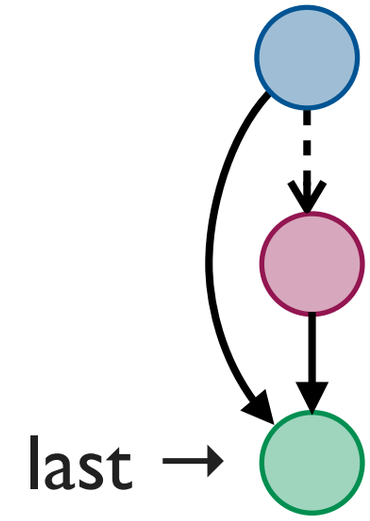
```
class MyClass {
  ...
  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
  }
}}
```



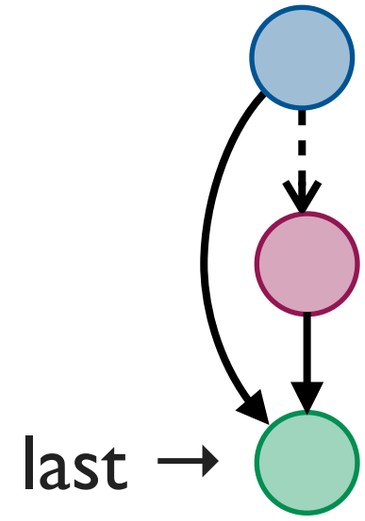
```
class MyClass {
  ...
  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
    }
}}
```



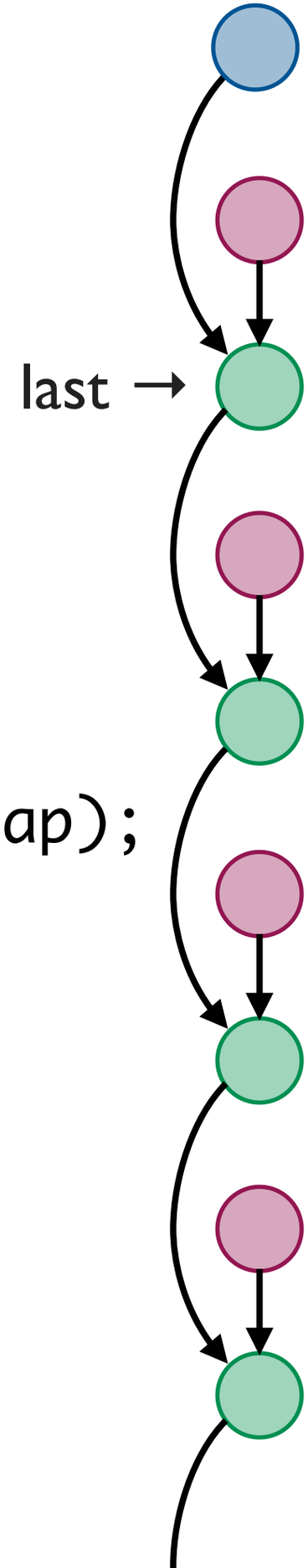
```
class MyClass {
  ...
  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
    }
}}
```



```
class MyClass {
  ...
  task doCompute(Vector in) {
    Activation last = now;
    for(Object o : in) {
      Activation map = sched this.doMap(o);
      Activation write = sched this.doWrite(map);
      map → write;
      last → write;
      last = write;
  }
}}
```



```
class MyClass {

  ...

  task doCompute(Vector in) {

    Activation last = now;

    for(Object o : in) {

      Activation map = sched this.doMap(o);

      Activation write = sched this.doWrite(map);

      map → write;

      last → write;

      last = write;

    }

}}
```



Other Optimizations

- Dependence reduction
- Removing lock-guards in intervals
- Removing happens-before assertions in intervals
- More?