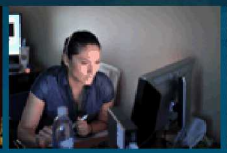
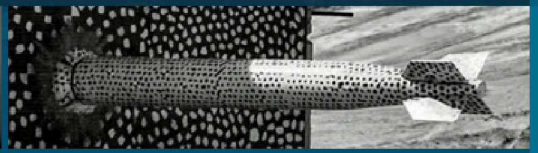
SAND2019-11249C

A Gentle Introduction to Java's New Memory Model





PRESENTED BY

John Bender, Jens Palsberg (UCLA)

This research was conducted as a graduate student at UCLA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

C11

```
...
let sw = [(REL | ACQ_REL | SC)];
  ([F]; sb)?; rs; rf;
  [R & (RLX | REL | ACQ | ACQ_REL | SC)];
  (sb; [F])?;
  [(ACQ | ACQ_REL | SC)]
...
```

C11

```
...
let sw = [(REL | ACQ_REL | SC)];
  ([F]; sb)?; rs; rf;
  [R & (RLX | REL | ACQ | ACQ_REL | SC)];
  (sb; [F])?;
  [(ACQ | ACQ_REL | SC)]
...
```

C11

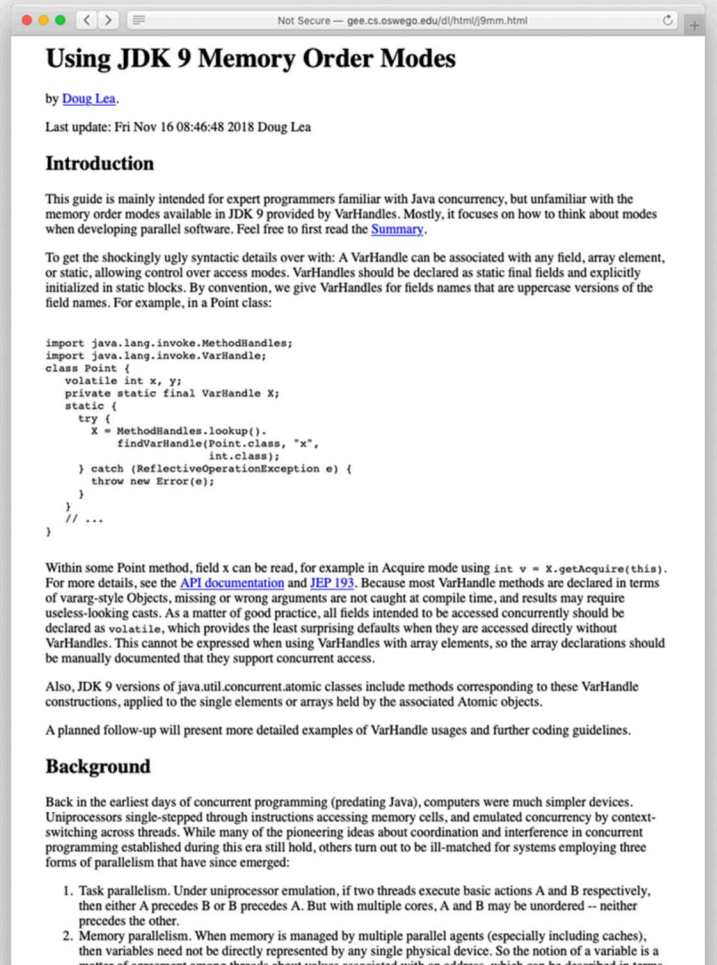
```
...
let sw = [(REL | ACQ_REL | SC)];
  ([F]; sb)?; rs; rf;
  [R & (RLX | REL | ACQ | ACQ_REL | SC)];
  (sb; [F])?;
  [(ACQ | ACQ_REL | SC)]
...
```

C11

```
...
let sw = [(REL | ACQ_REL | SC)];
  ([F]; sb)?; rs; rf;
  [R & (RLX | REL | ACQ | ACQ_REL | SC)];
  (sb; [F])?;
  [(ACQ | ACQ_REL | SC)]
...
```

20

Informal Specification



Not Secure — gee.cs.oswego.edu/dl/html/j9mm.html

Using JDK 9 Memory Order Modes

by Doug Lea.

Last update: Fri Nov 16 08:46:48 2018 Doug Lea

Introduction

This guide is mainly intended for expert programmers familiar with Java concurrency, but unfamiliar with the memory order modes available in JDK 9 provided by VarHandles. Mostly, it focuses on how to think about modes when developing parallel software. Feel free to first read the Summary.

To get the shockingly ugly syntactic details over with: A VarHandle can be associated with any field, array element, or static, allowing control over access modes. VarHandles should be declared as static final fields and explicitly initialized in static blocks. By convention, we give VarHandles for fields names that are uppercase versions of the field names. For example, in a Point class:

```
import java.lang.invoke.MethodHandles;
import java.lang.invoke.VarHandle;
class Point {
   volatile int x, y;
   private static final VarHandle X;
   static {
     try {
       X = MethodHandles.lookup().
           findVarHandle(Point.class, "x",
                         int.class);
     } catch (ReflectiveOperationException e) {
       throw new Error(e);
     }
   }
   // ...
}
```

Within some Point method, field x can be read, for example in Acquire mode using `int v = X.getAcquire(this)`. For more details, see the API documentation and JEP 193. Because most VarHandle methods are declared in terms of vararg-style Objects, missing or wrong arguments are not caught at compile time, and results may require useless-looking casts. As a matter of good practice, all fields intended to be accessed concurrently should be declared as `volatile`, which provides the least surprising defaults when they are accessed directly without VarHandles. This cannot be expressed when using VarHandles with array elements, so the array declarations should be manually documented that they support concurrent access.

Also, JDK 9 versions of java.util.concurrent.atomic classes include methods corresponding to these VarHandle constructions, applied to the single elements or arrays held by the associated Atomic objects.

A planned follow-up will present more detailed examples of VarHandle usages and further coding guidelines.

Background

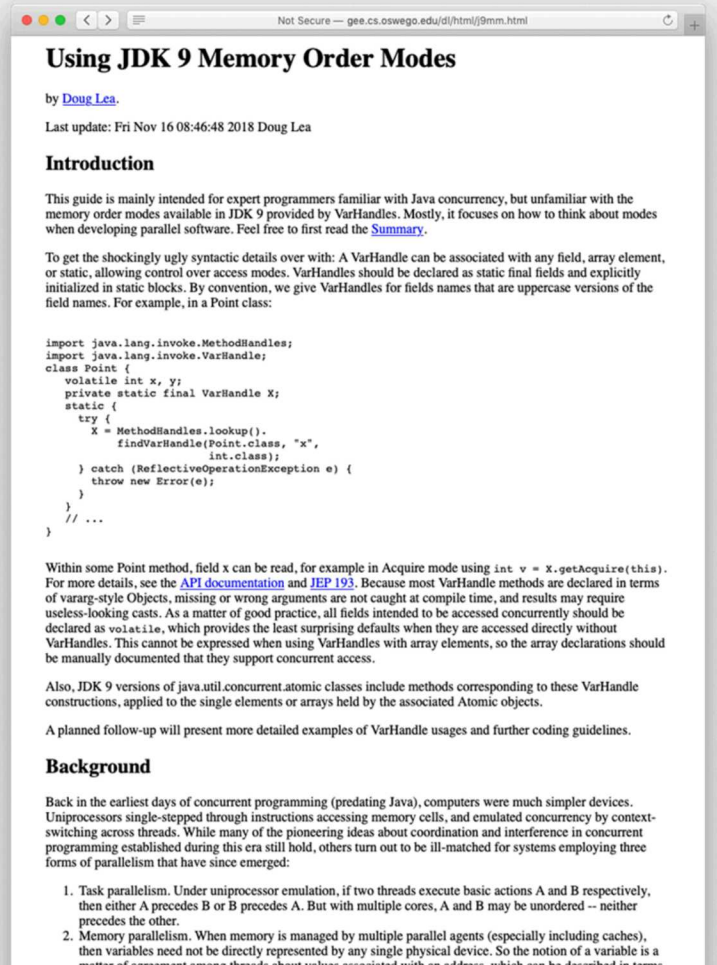
Back in the earliest days of concurrent programming (predating Java), computers were much simpler devices. Uniprocessors single-stepped through instructions accessing memory cells, and emulated concurrency by context-switching across threads. While many of the pioneering ideas about coordination and interference in concurrent programming established during this era still hold, others turn out to be ill-matched for systems employing three forms of parallelism that have since emerged:

1. Task parallelism. Under uniprocessor emulation, if two threads execute basic actions A and B respectively, then either A precedes B or B precedes A. But with multiple cores, A and B may be unordered -- neither precedes the other.
2. Memory parallelism. When memory is managed by multiple parallel agents (especially including caches), then variables need not be directly represented by any single physical device. So the notion of a variable is a

Informal Specification

Platform Specific

Architecture assumptions



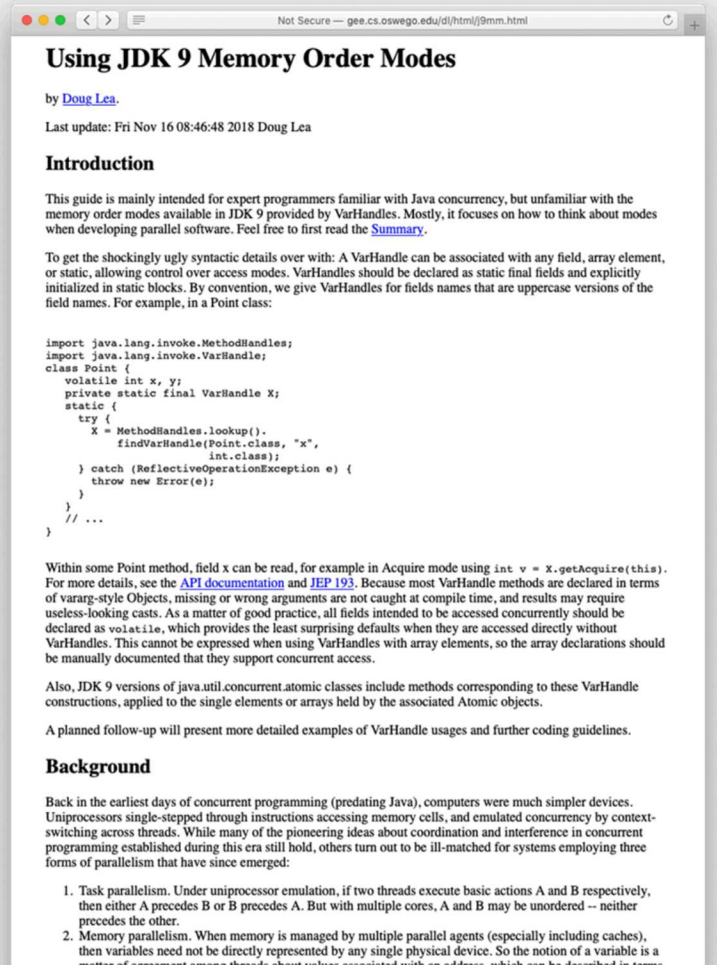
Informal Specification

Platform Specific

Architecture assumptions

Incorrect

Undersynchronized



Informal Specification

Platform Specific

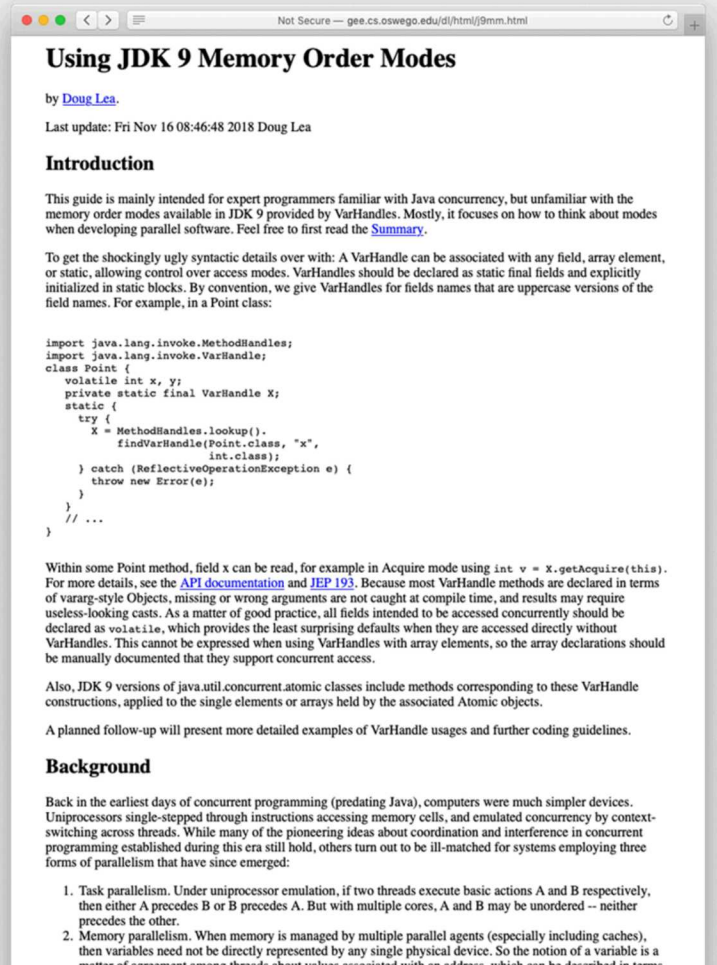
Architecture assumptions

Incorrect

Undersynchronized

Slow

Oversynchronized



Not Secure — gee.cs.oswego.edu/dl/html/j9mm.html

Using JDK 9 Memory Order Modes

by Doug Lea.

Last update: Fri Nov 16 08:46:48 2018 Doug Lea

Introduction

This guide is mainly intended for expert programmers familiar with Java concurrency, but unfamiliar with the memory order modes available in JDK 9 provided by VarHandles. Mostly, it focuses on how to think about modes when developing parallel software. Feel free to first read the Summary.

To get the shockingly ugly syntactic details over with: A VarHandle can be associated with any field, array element, or static, allowing control over access modes. VarHandles should be declared as static final fields and explicitly initialized in static blocks. By convention, we give VarHandles for fields names that are uppercase versions of the field names. For example, in a Point class:

```
import java.lang.invoke.MethodHandles;
import java.lang.invoke.VarHandle;
class Point {
   volatile int x, y;
   private static final VarHandle X;
   static {
     try {
       X = MethodHandles.lookup().
           findVarHandle(Point.class, "x",
                         int.class);
     } catch (ReflectiveOperationException e) {
       throw new Error(e);
     }
   }
   // ...
}
```

Within some Point method, field x can be read, for example in Acquire mode using `int v = X.getAcquire(this)`. For more details, see the API documentation and JEP 193. Because most VarHandle methods are declared in terms of vararg-style Objects, missing or wrong arguments are not caught at compile time, and results may require useless-looking casts. As a matter of good practice, all fields intended to be accessed concurrently should be declared as `volatile`, which provides the least surprising defaults when they are accessed directly without VarHandles. This cannot be expressed when using VarHandles with array elements, so the array declarations should be manually documented that they support concurrent access.

Also, JDK 9 versions of java.util.concurrent.atomic classes include methods corresponding to these VarHandle constructions, applied to the single elements or arrays held by the associated Atomic objects.

A planned follow-up will present more detailed examples of VarHandle usages and further coding guidelines.

Background

Back in the earliest days of concurrent programming (predating Java), computers were much simpler devices. Uniprocessors single-stepped through instructions accessing memory cells, and emulated concurrency by context-switching across threads. While many of the pioneering ideas about coordination and interference in concurrent programming established during this era still hold, others turn out to be ill-matched for systems employing three forms of parallelism that have since emerged:

1. Task parallelism. Under uniprocessor emulation, if two threads execute basic actions A and B respectively, then either A precedes B or B precedes A. But with multiple cores, A and B may be unordered -- neither precedes the other.
2. Memory parallelism. When memory is managed by multiple parallel agents (especially including caches), then variables need not be directly represented by any single physical device. So the notion of a variable is a

Informal Specification

Platform Specific

Architecture assumptions

Incorrect

Undersynchronized

Slow

Oversynchronized

```
...
let sw = [(REL | ACQ_REL | SC)];
  ([F]; sb)?; rs; rf;
  [R & (RLX | REL | ACQ | ACQ_REL | SC)];
  (sb; [F])?;
  [(ACQ | ACQ_REL | SC)]
...
```

Informal Specification

Platform Specific

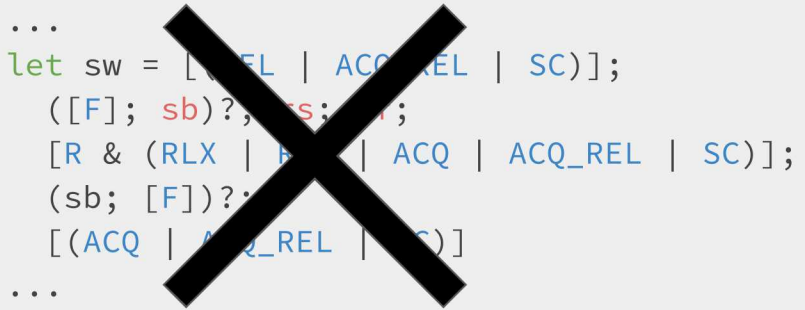
Architecture assumptions

Incorrect

Undersynchronized

Slow

Oversynchronized



What is a memory model for?

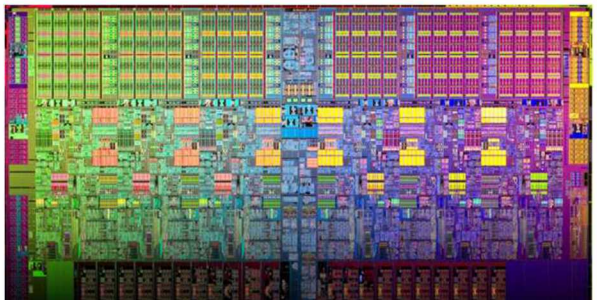
What values can a read see?

v1. What values can a read see?

Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

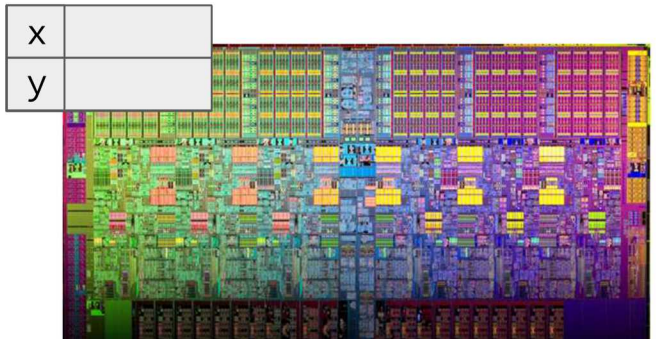
```
thread1:
read x
read y
```



Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

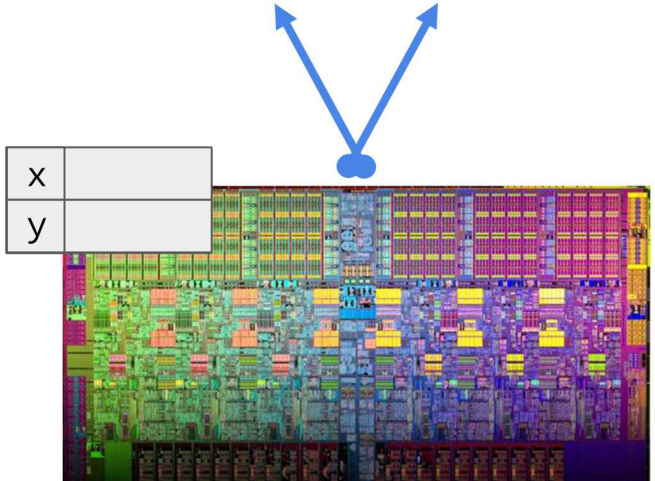
```
thread1:
read x
read y
```



Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

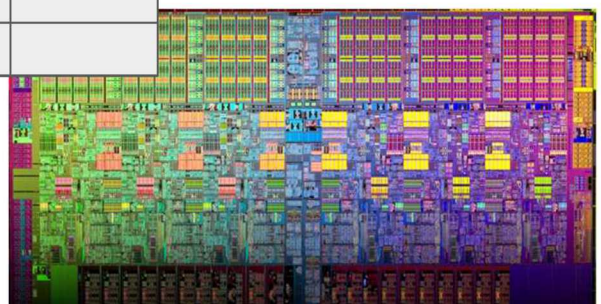
```
thread1:
read x
read y
```



Sequential Consistency

```
thread0:                thread1:
write 1 to y            read x
write 1 to x            read y
```

x	
y	

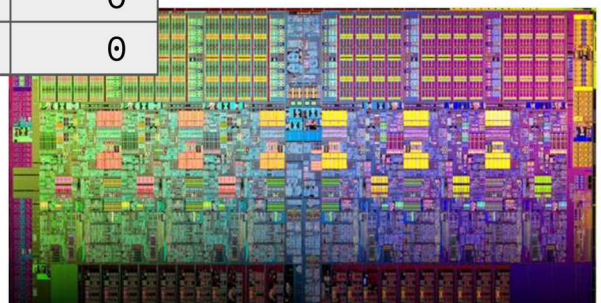


Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

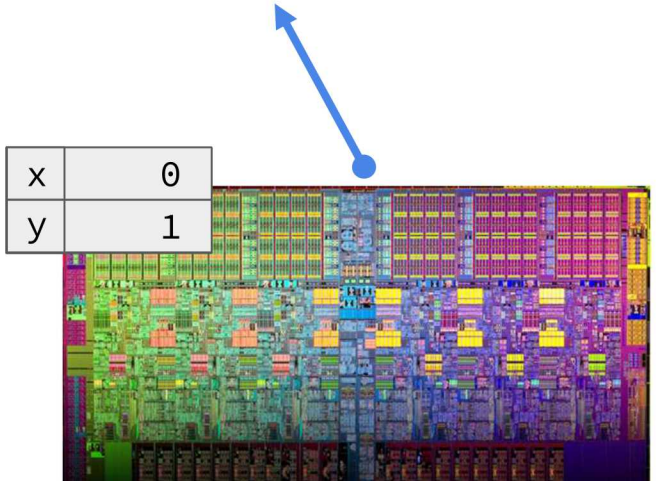
```
thread1:
read x
read y
```

x	0
y	0



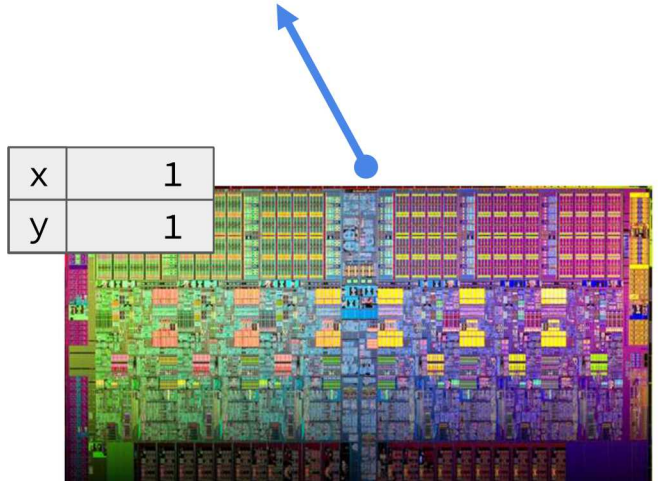
Sequential Consistency

```
thread0:                thread1:
▶ write 1 to y          read x
  write 1 to x          read y
```



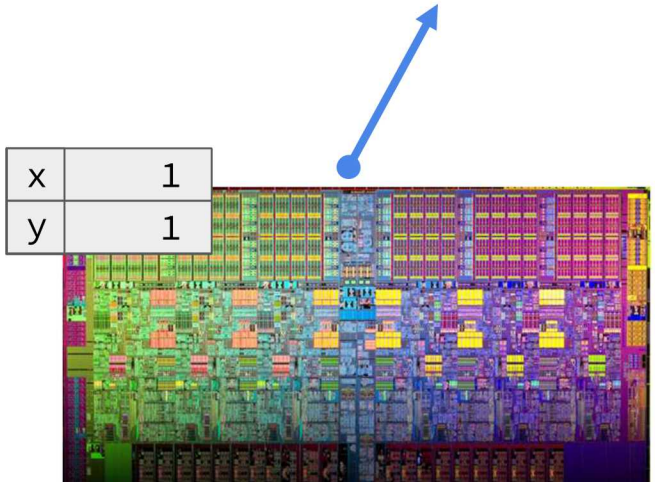
Sequential Consistency

```
thread0:                thread1:
write 1 to y            read x
▶ write 1 to x          read y
```



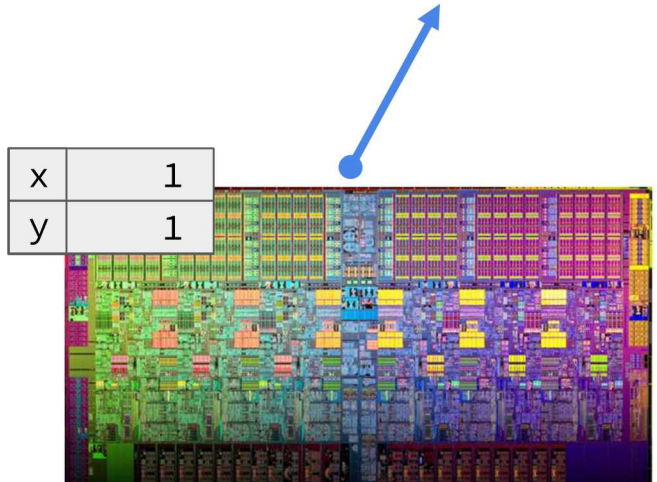
Sequential Consistency

```
thread0:                thread1:
write 1 to y          ▶ read x // 1
▶ write 1 to x          read y
```



Sequential Consistency

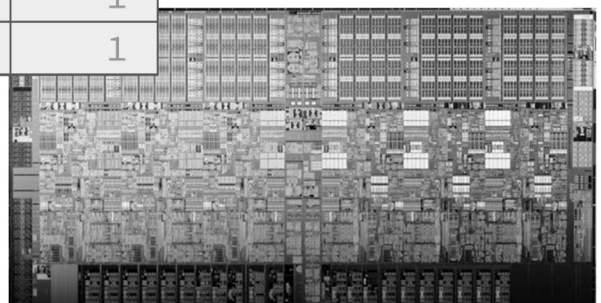
```
thread0:                thread1:
write 1 to y            read x // 1
▶ write 1 to x          ▶ read y // 1
```



```
thread0:                thread1:
write 1 to y            read x // 1
▶ write 1 to x          ▶ read y // 1
```

v1. What values can a read see?

x	1
y	1



Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y
```

Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y
```

1. Linear order of execution

Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y
```

1. Linear order of execution
2. Program order consistent

Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y
```

1. Linear order of execution
2. Program order consistent
3. Reads from last write

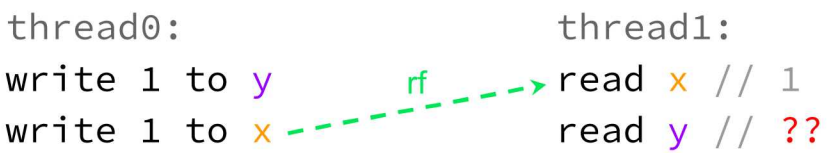
Sequential Consistency

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x // 1
read y // ??
```

1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Consistency

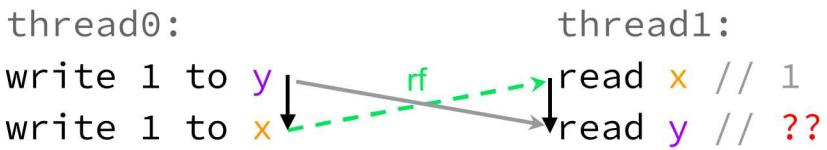


1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Consistency

1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Consistency



1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Consistency

1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Consistency

1. Linear order of execution
2. Program order consistent
3. ▶ Reads from last write

v1. What values can a read see?

v2. Which write is paired with a read?

v2. Which write is paired with a read?

```
init:
write 0 to y
```

v2. Which write is paired with a read?

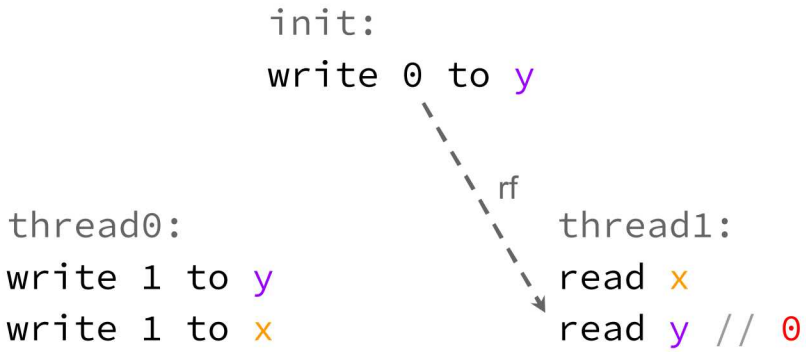
Write elimination

```
init:
write 0 to y
```

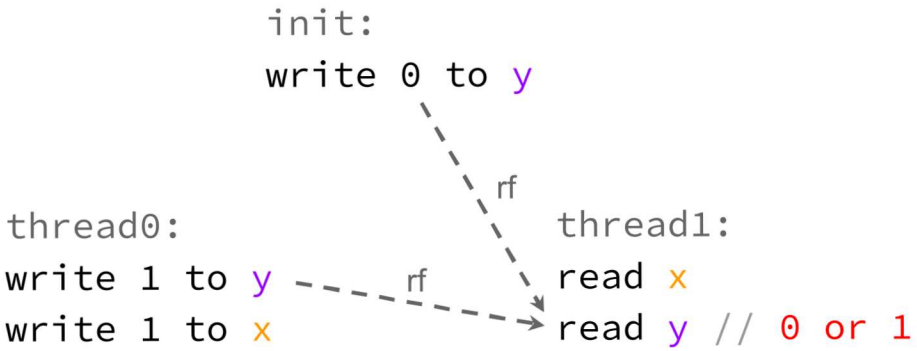
```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y // ??
```

Write elimination

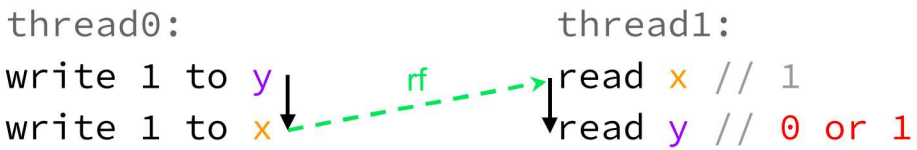


Write elimination



Write elimination

```
init:
write 0 to y
```



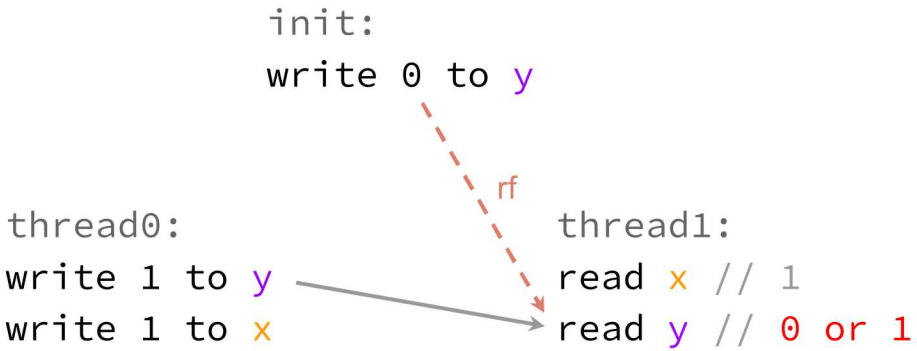
Write elimination

```
init:
write 0 to y
```

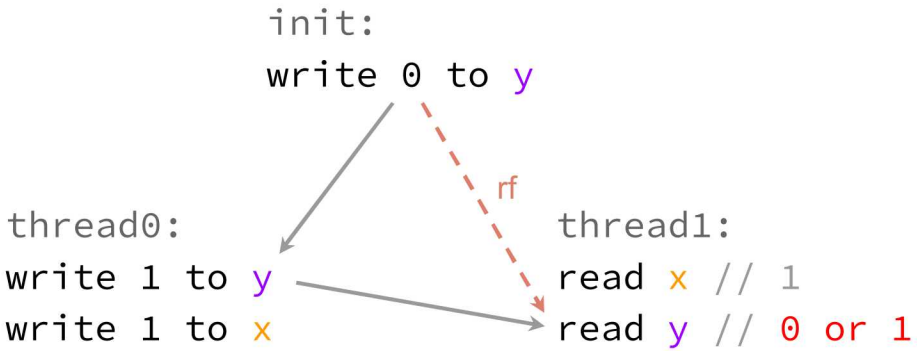
```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x // 1
read y // 0 or 1
```

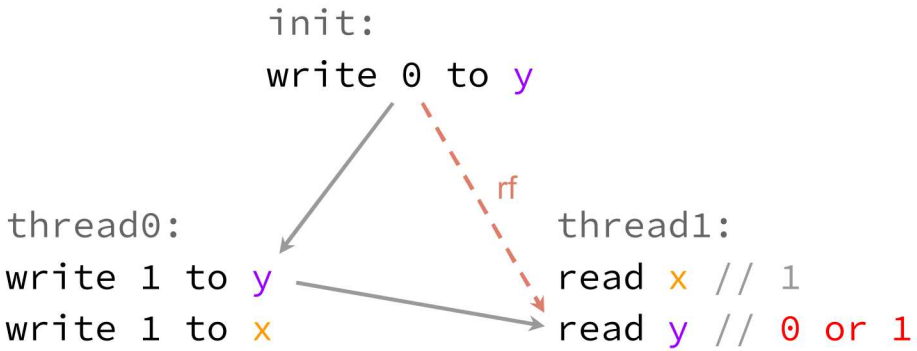
Write elimination



Write elimination

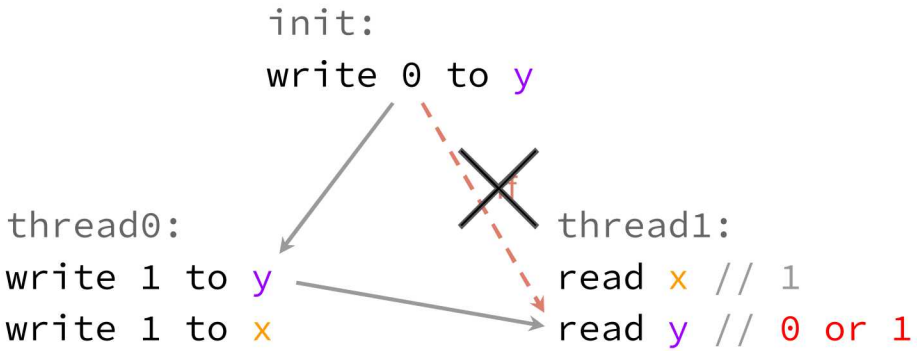


Write elimination



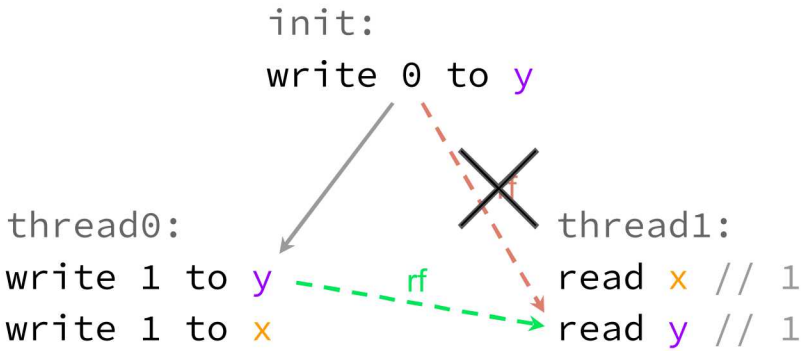
1. Linear order of execution
2. Program order consistent
3. Reads from last write

Write elimination



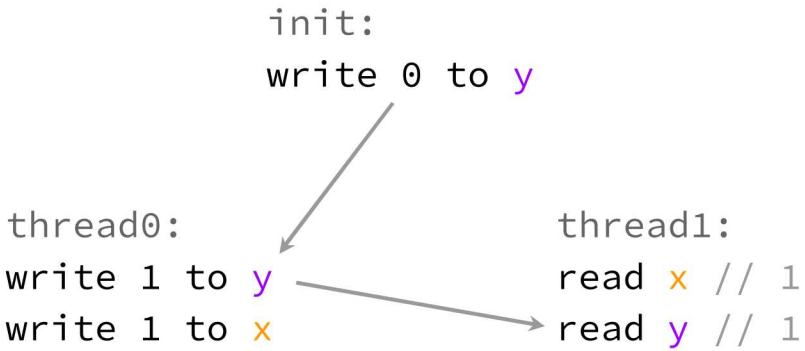
1. Linear order of execution
2. Program order consistent
3. Reads from last write

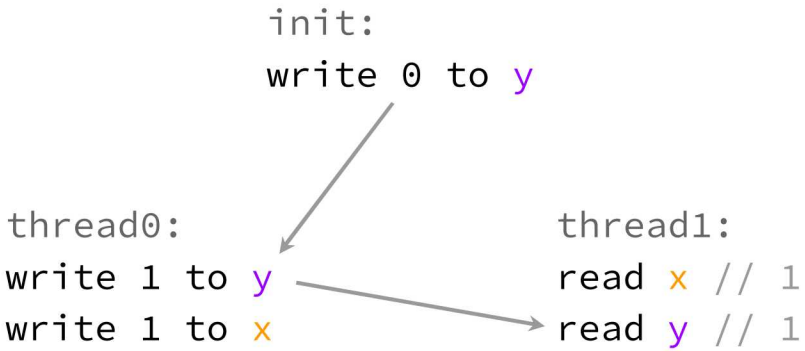
Write elimination



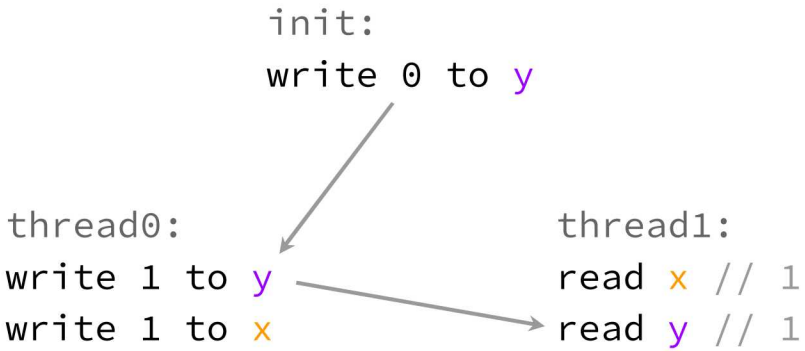
1. Linear order of execution
2. Program order consistent
3. Reads from last write

Write elimination

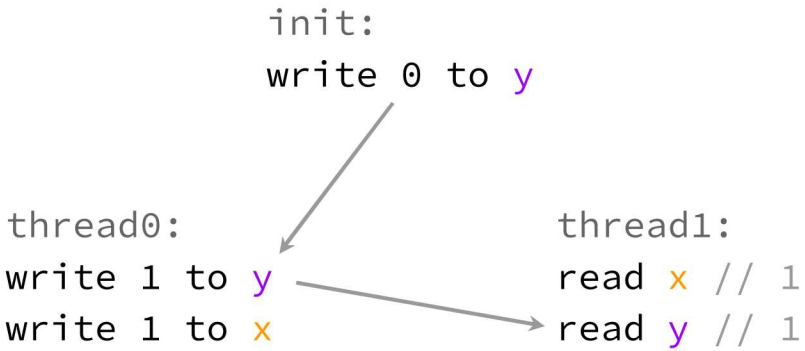




v2. Which write is paired with a read?



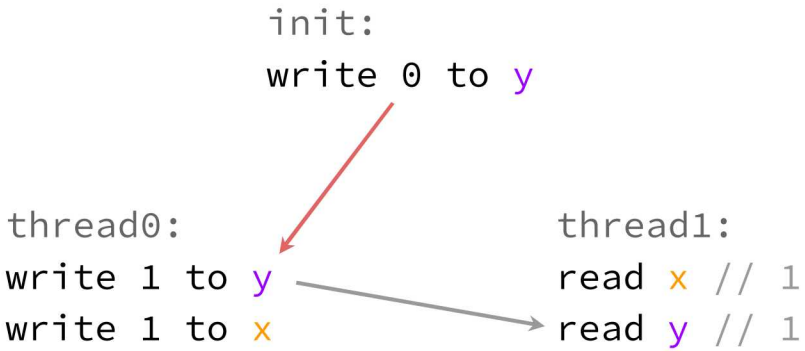
v3. When are access effects visible?



v3. When are access effects visible?

to reads?

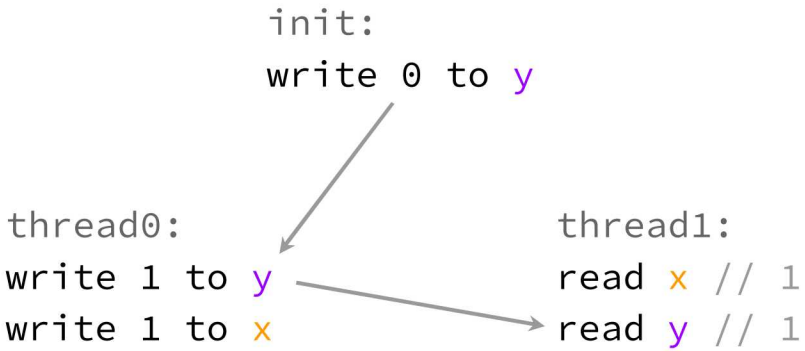
Write elimination



v3. When are access effects visible?

to reads?

Write elimination



SC

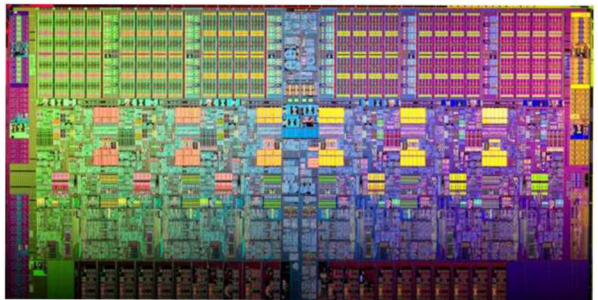
1. Linear order of execution
2. Program order consistent
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x // 1
read y // ??
```

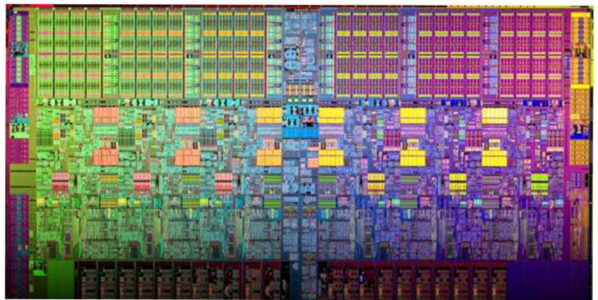


Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x // 1
read y // ??
```

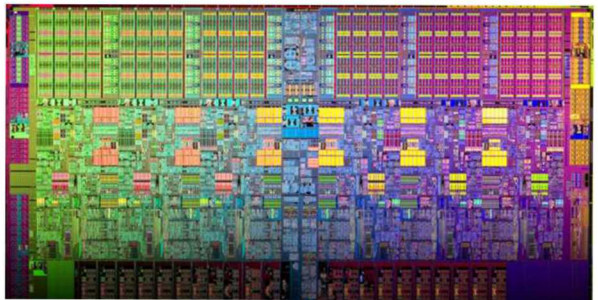


Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to x
write 1 to y
```

```
thread1:
read x // 1
read y // ??
```



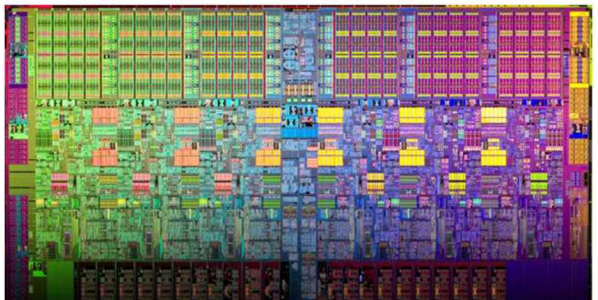
Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to x
write 1 to y
```

```
thread1:
read x // 1
read y // ??
```

1. Linear order of execution
2. Program order consistent
3. Reads from last write



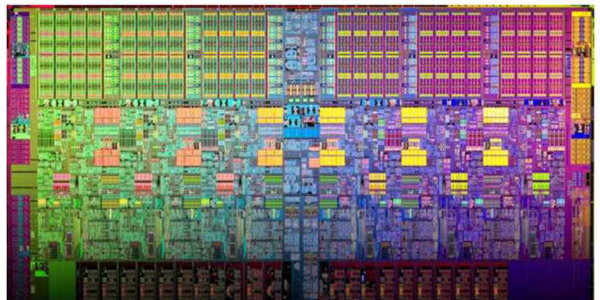
Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to x
write 1 to y
```

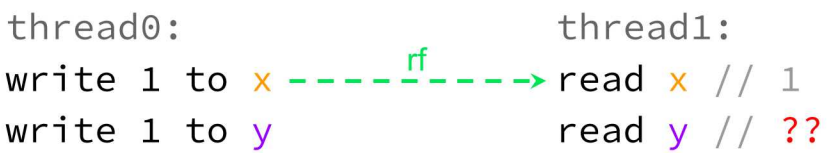
```
thread1:
read x // 1
read y // ??
```

1. Linear order of execution
2. ~~Program order consistent~~
3. Reads from last write



Sequential Inconsistency

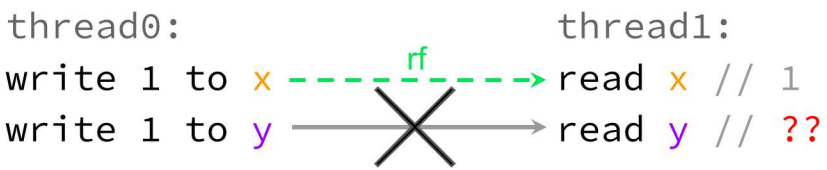
```
init:
write 0 to y
```



1. Linear order of execution
2. ~~Program order consistent~~
3. Reads from last write

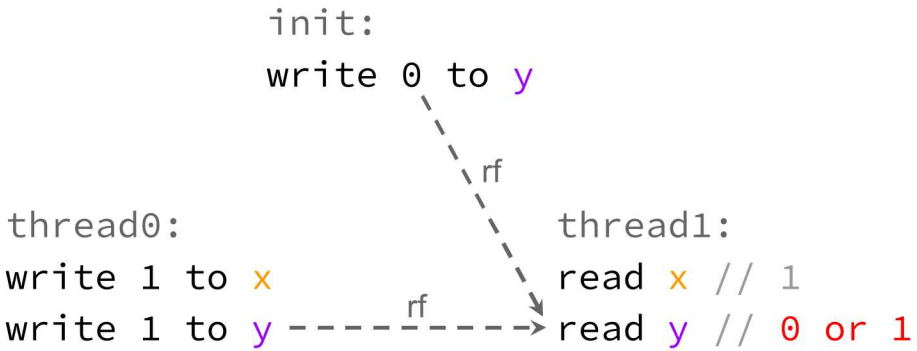
Sequential Inconsistency

```
init:
write 0 to y
```



1. Linear order of execution
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency



1. Linear order of execution
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x
read y // ??
```

1. ~~Linear order of execution~~
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:                    thread1:
write 1 to y      rf        read x // 1
write 1 to x  - - - - - ->  read y // ??
```

1. ~~Linear order of execution~~
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

1. ~~Linear order of execution~~
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:
write 1 to y
write 1 to x
```


```
thread1:
read x // 1
read y // ??
```

1. ~~Linear order of execution~~
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

```
thread0:                    thread1:
write 1 to y      ?         read x // 1
write 1 to x                read y // ??
```

v3. When are access effects visible?

1. ~~Linear order of execution~~
2. ~~Program order consistent~~
3. Reads from last write

Sequential Inconsistency

```
init:
write 0 to y
```

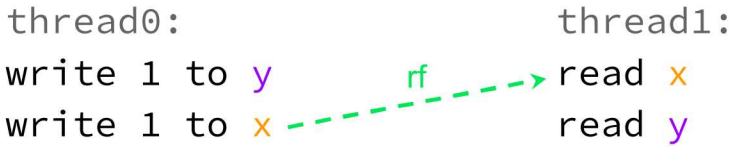
```
thread0:
write 1 to y
write 1 to x
```


```
thread1:
read x // 1
read y // ??
```

1. Linear order of execution
2. Program order consistent
3. Reads from last write

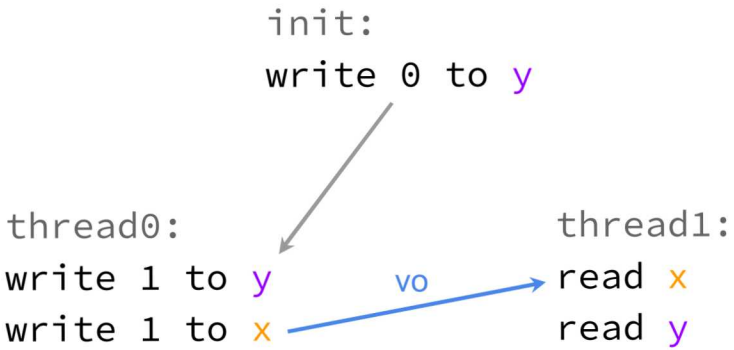
Visibility

```
init:
write 0 to y
```



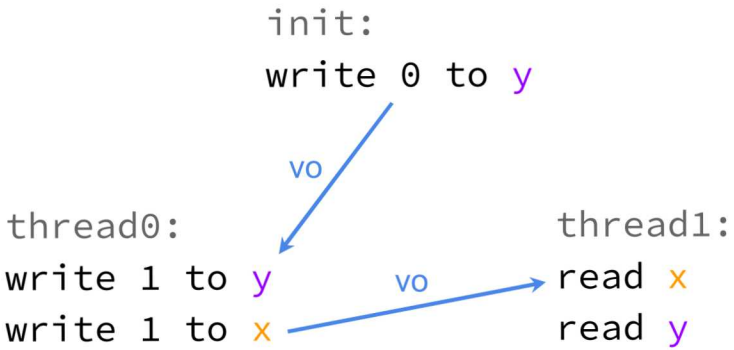
```
init:
write 0 to y
```

1. Reads


init:
write 0 to y
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x
read y

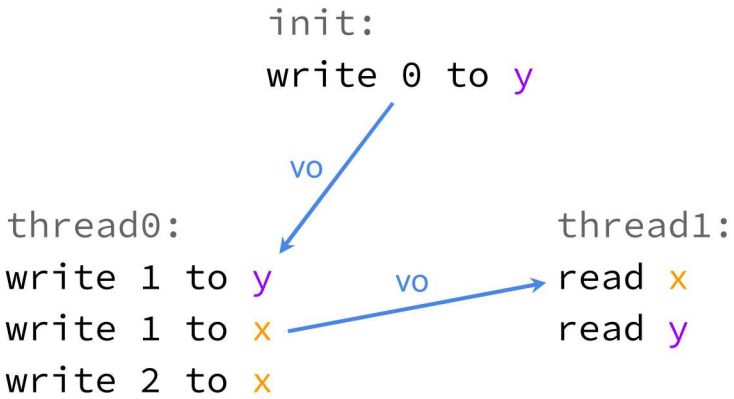
1. Reads

Visibility



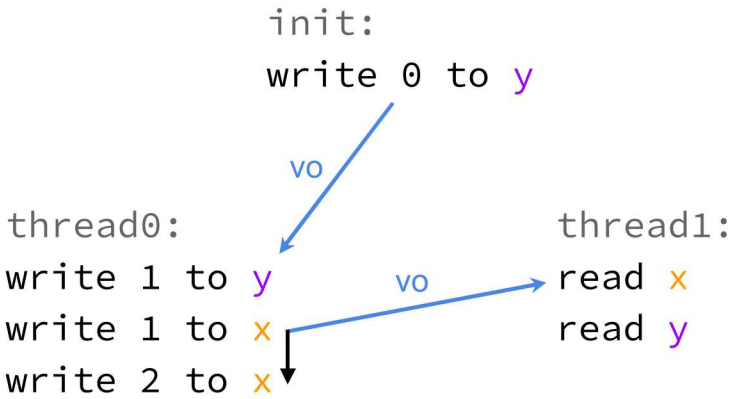
1. Reads
2. Initial writes

Visibility



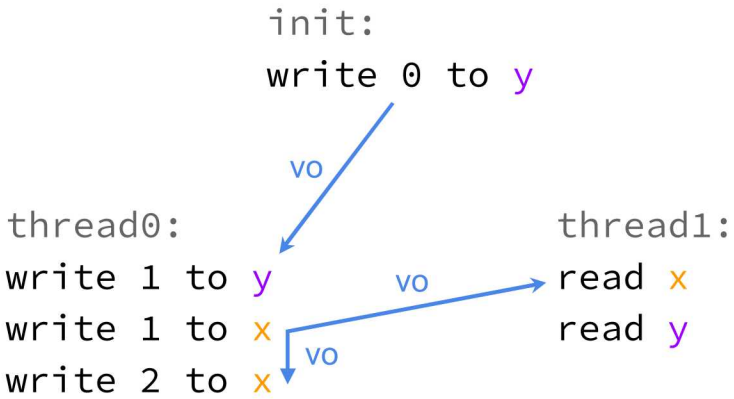
1. Reads
2. Initial writes

Visibility



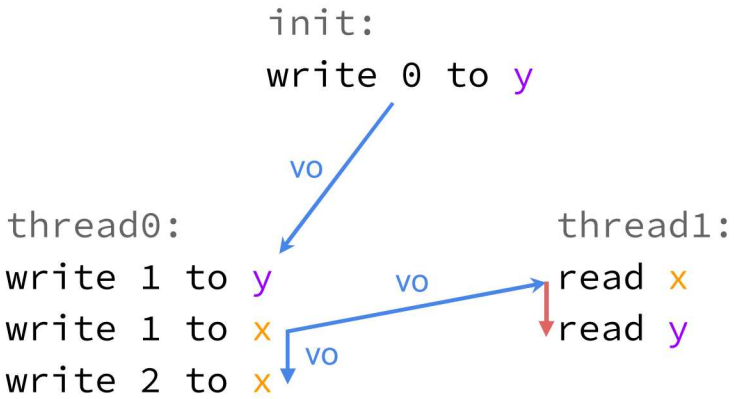
1. Reads
2. Initial writes

Visibility



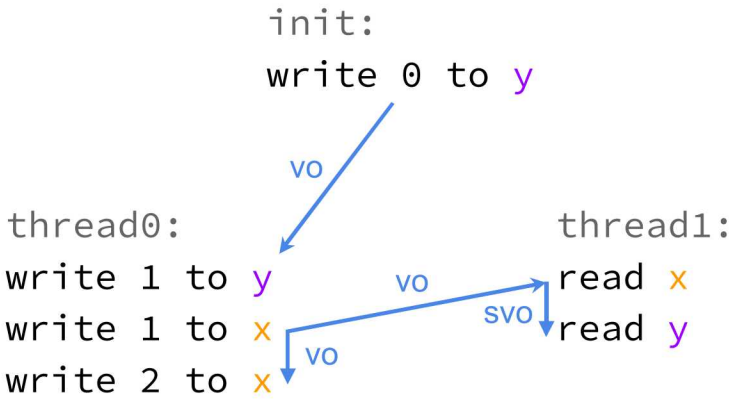
1. Reads
2. Initial writes
3. Same location

Visibility



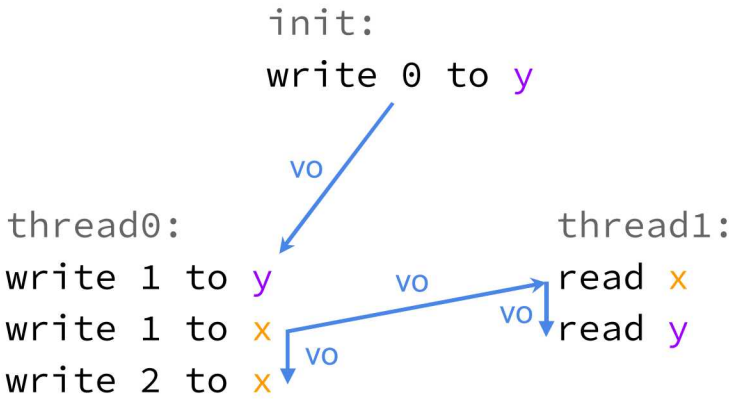
1. Reads
2. Initial writes
3. Same location

Visibility



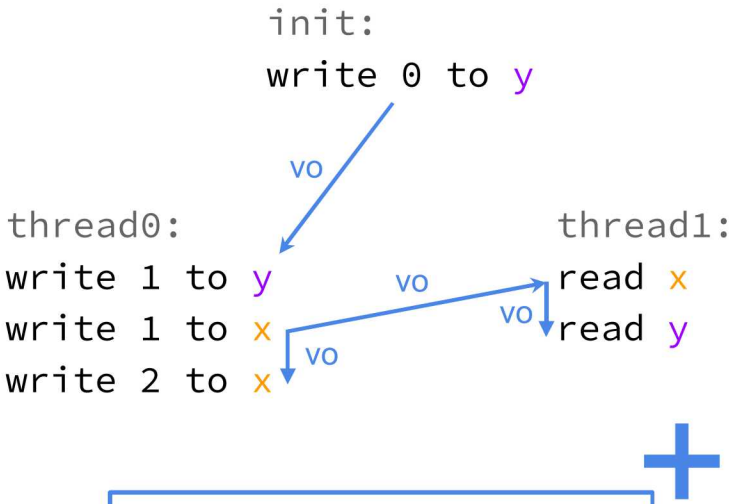
1. Reads
2. Initial writes
3. Same location

Visibility



1. Reads
2. Initial writes
3. Same location
4. Specified

Visibility



1. Reads
2. Initial writes
3. Same location
4. Specified

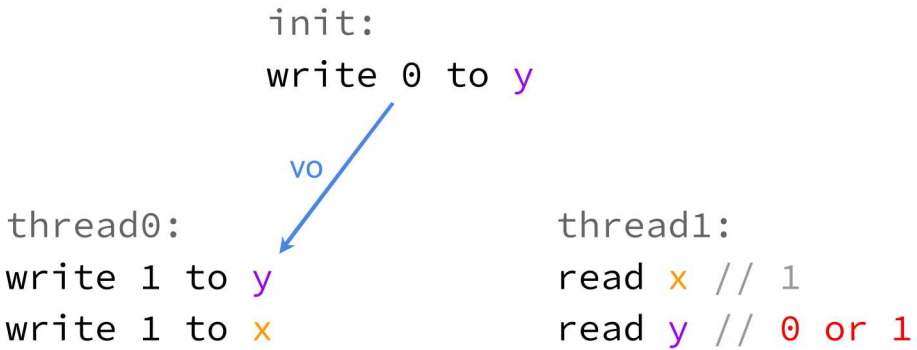
Visibility

```
init:
write 0 to y
```

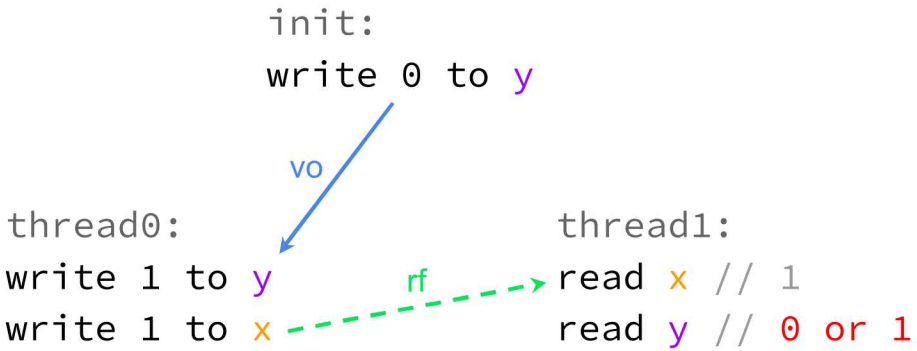
```
thread0:
write 1 to y
write 1 to x
```

```
thread1:
read x // 1
read y // 0 or 1
```

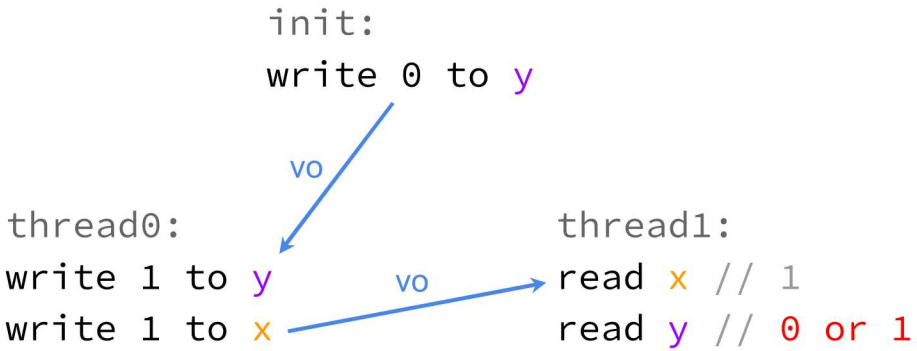
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
thread1:
read x // 1
read y // 0 or 1

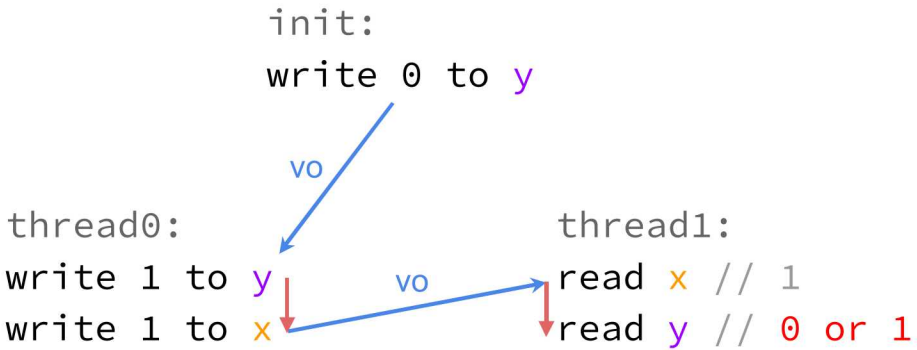
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
rf
thread1:
read x // 1
read y // 0 or 1

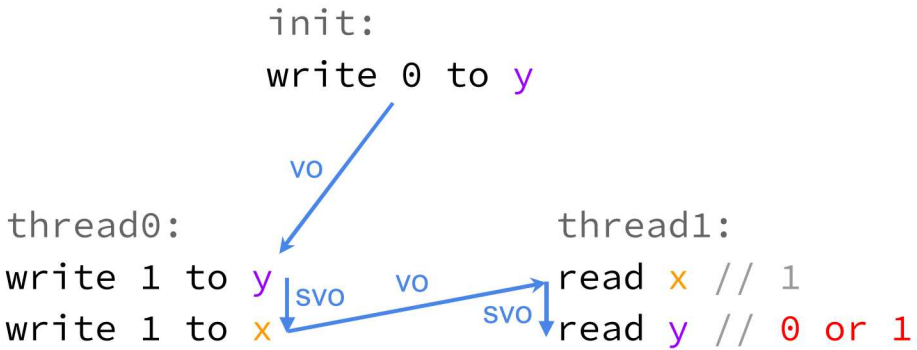
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1

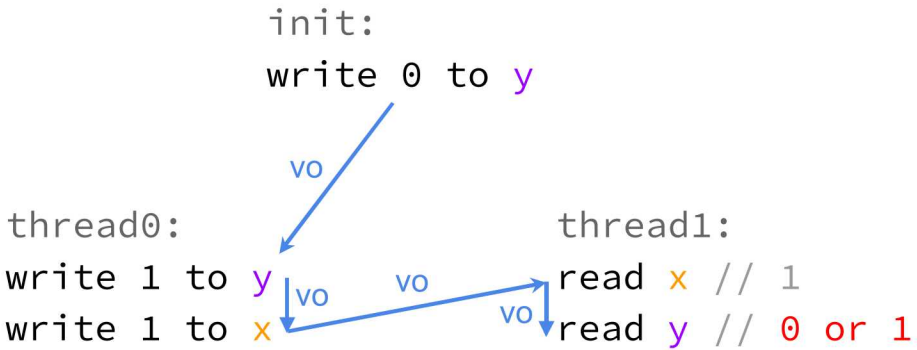
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1

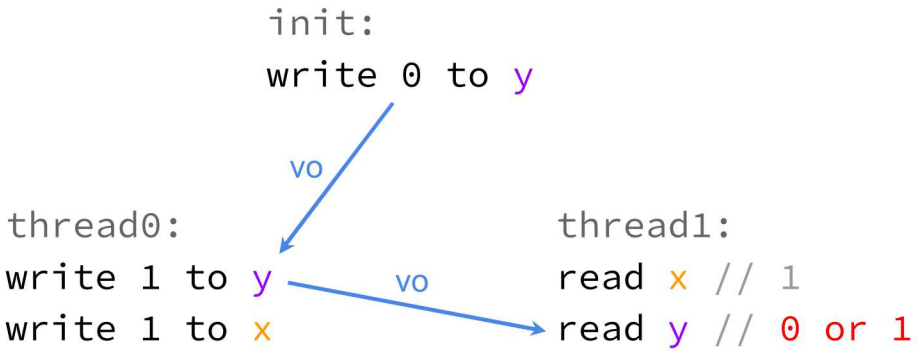
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
svo
vo
svo
thread1:
read x // 1
read y // 0 or 1

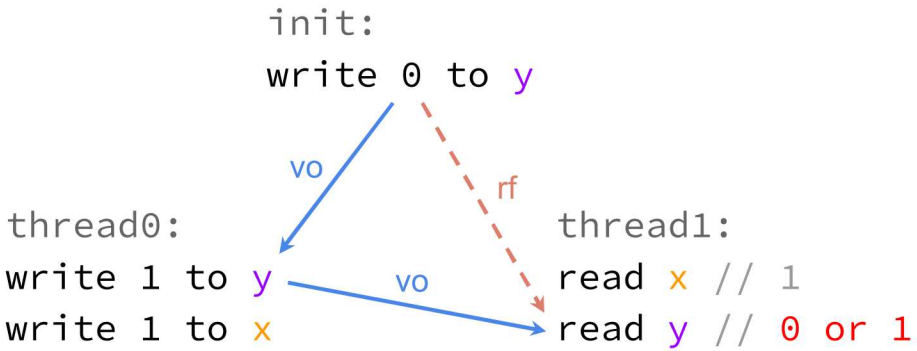
Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
vo
vo
vo
thread1:
read x // 1
read y // 0 or 1

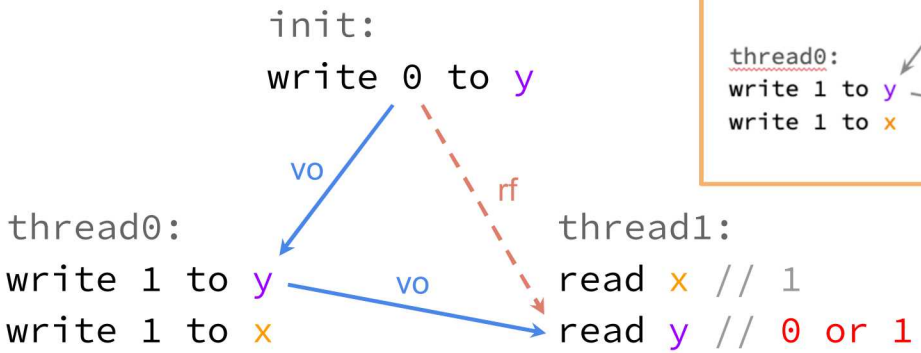
Visibility

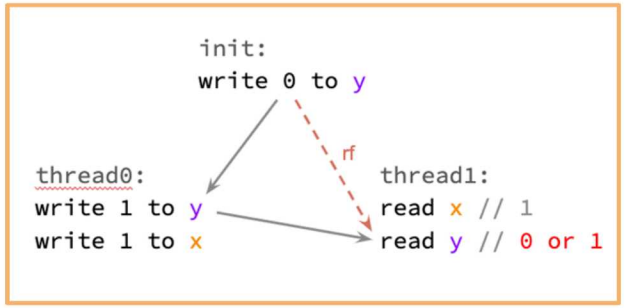

init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1

Visibility

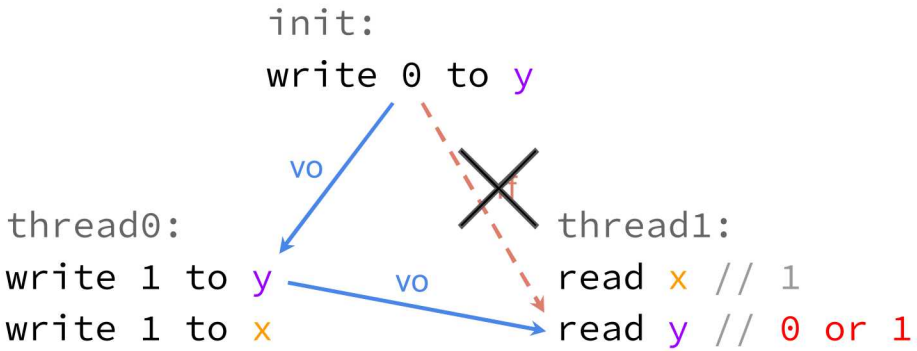

init:
write 0 to y
vo
rf
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1

Visibility

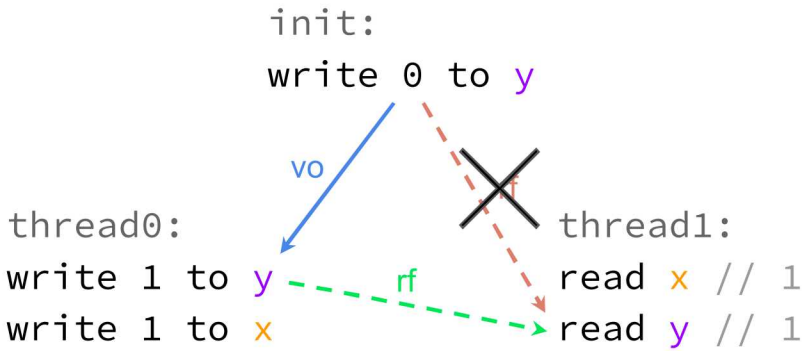

init:
write 0 to y
vo
rf
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1


init:
write 0 to y
rf
thread0:
write 1 to y
write 1 to x
thread1:
read x // 1
read y // 0 or 1

Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
vo
thread1:
read x // 1
read y // 0 or 1

Visibility


init:
write 0 to y
vo
thread0:
write 1 to y
write 1 to x
rf
thread1:
read x // 1
read y // 1

VarHandle API

VarHandle API

```
read y
write 1 to y
```

VarHandle API

Plain

Minimal guarantees

```
read y
write 1 to y

a = y;
y = 1;
```

OOPSLA'19

VarHandle API

Plain

Minimal guarantees

Opaque

Acyclic causality

```
read y
write 1 to y

a = y;
y = 1;

Y.getOpaque();
Y.setOpaque(1);
```

OOPSLA'19

VarHandle API

Plain

Minimal guarantees

Opaque

Acyclic causality

Release-acquire

Message passing

```
read y
write 1 to y

a = y;
y = 1;

Y.getOpaque();
Y.setOpaque(1);

Y.getAcquire();
Y.setRelease(1);
```

VarHandle API

Plain

Minimal guarantees

Opaque

Acyclic causality

Release-acquire

Message passing

Volatile

SC semantics, `volatile` variables

```
read y
write 1 to y

a = y;
y = 1;

Y.getOpaque();
Y.setOpaque(1);

Y.getAcquire();
Y.setRelease(1);

Y.getVolatile();
Y.setVolatile(1);
```

VarHandle API

Plain

Minimal guarantees

Opaque

Acyclic causality

Release-acquire

Message passing

Volatile

SC semantics, `volatile` variables

```
read y
write 1 to y

a = y;
y = 1;

Y.getOpaque();
Y.setOpaque(1);

Y.getAcquire();
Y.setRelease(1);

Y.getVolatile();
Y.setVolatile(1);
```

OOPSLA'19

VarHandle API

Plain

Minimal guarantees

Opaque

Acyclic causality

Release-acquire

Message passing

Volatile

SC semantics, `volatile` variables

plain ⊑ opaque ⊑ release-acquire ⊑ volatile

```
read y
write 1 to y

a = y;
y = 1;

Y.getAcquire();
Y.setRelease(1);

Y.getVolatile();
Y.setVolatile(1);
```

Model

OOPSLA ‘19

A Formalization of Java’s Concurrent Access Modes

ANONYMOUS AUTHOR(S)

Java’s memory model was recently updated and expanded with new access modes. The accompanying documentation for these access modes is intended to make strong guarantees about program behavior that the Java compiler must enforce, yet the documentation is frequently unclear. This makes the intended program behavior ambiguous, impedes discussion of key design decisions, and makes it impossible to prove general properties about the semantics of the access modes.

In this paper we present the first formalization of Java’s access modes. We have constructed an axiomatic model for all of the modes using the Herd modeling tool. This allows us to give precise answers to questions about the behavior of example programs, called litmus tests. We have validated our model using a large suite of litmus tests from existing research which helps to shed light on the relationship with other memory models. We have also modeled the semantics in Coq and proven several general theorems including a DRF guarantee, which says that if a program is properly synchronized then it will exhibit sequentially consistent behavior. Finally, we use our model to prove that the unusual design choice of a partial order among writes to the same location is unobservable in any program.

1 INTRODUCTION

The original Java memory model [Manson et al. 2005] included an early attempt to define the semantics of lock-free shared memory programs running on the Java platform, but the definitions were hard to understand and there was no easy way to check the behavior of example programs. It was also later discovered that it ruled out existing compiler optimizations which it claimed to support [Ševčík and Aspinall 2008]. Since then, researchers have made great advances in memory model design while studying other weak memory models like those for ARM [Alglave et al. 2008; Pulte et al. 2017], C11 [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017; Vafeiadis et al. 2015], Power [Alglave et al. 2014], and x86 [Owens et al. 2009].

Recently, the ninth version of the Java Development Kit updated and expanded Java’s memory model using new "access modes". Though the design of the access modes is inspired by C11’s memory orders [Committee et al. 2010], it differs in a few key ways. First, it sheds complicated legacy features like release sequences and release-consume accesses. Second, it includes a broad but simple mechanism to forbid so called “out of thin-air” behavior [Batty and Sewell 2014]. Finally, it makes no provision for a total order on writes to the same location. Taken together this suggests new opportunities to use a simpler model, develop metatheory, and verify lock-free algorithms for the Java platform.

However, the documentation [JDK9 2017; Lea 2017, 2018] is frequently ambiguous. This makes it extremely difficult to provide definitive answers about program behavior and there is little hope of proving important properties about the semantics. Further, it impedes the discussion of key features of the model’s design.

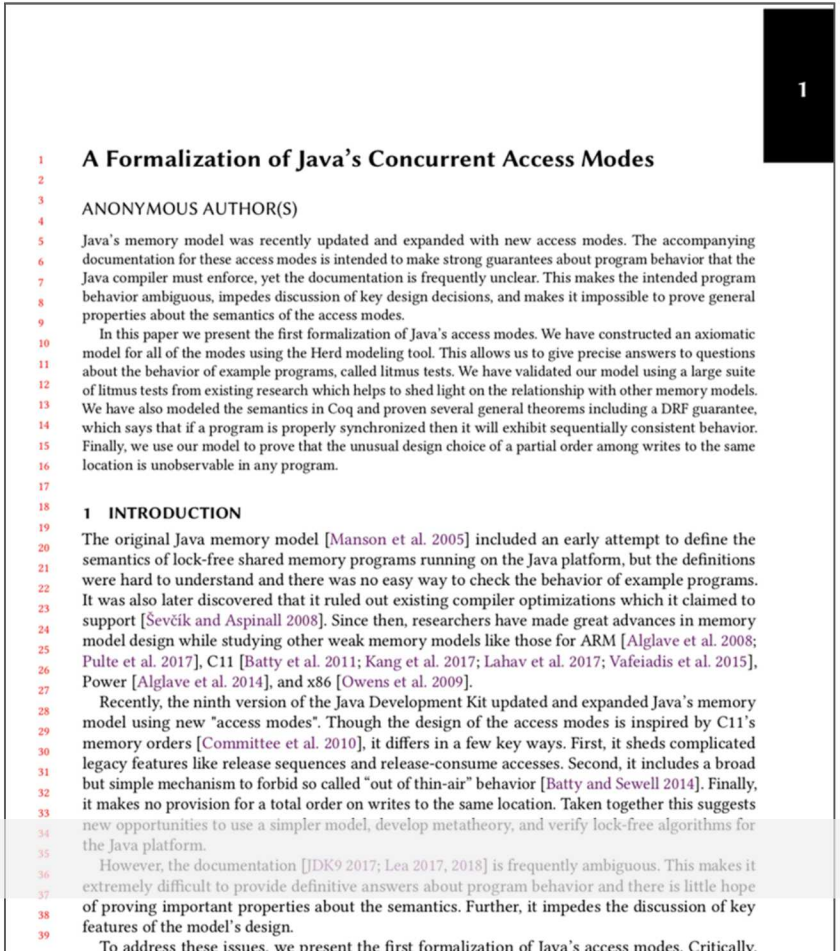
To address these issues, we present the first formalization of Java’s access modes. Critically,

Model

OOPSLA ‘19

Java Access Modes (JAM) Model

Instantiations in Herd[1] and Coq



A Formalization of Java’s Concurrent Access Modes

ANONYMOUS AUTHOR(S)

Java’s memory model was recently updated and expanded with new access modes. The accompanying documentation for these access modes is intended to make strong guarantees about program behavior that the Java compiler must enforce, yet the documentation is frequently unclear. This makes the intended program behavior ambiguous, impedes discussion of key design decisions, and makes it impossible to prove general properties about the semantics of the access modes.

In this paper we present the first formalization of Java’s access modes. We have constructed an axiomatic model for all of the modes using the Herd modeling tool. This allows us to give precise answers to questions about the behavior of example programs, called litmus tests. We have validated our model using a large suite of litmus tests from existing research which helps to shed light on the relationship with other memory models. We have also modeled the semantics in Coq and proven several general theorems including a DRF guarantee, which says that if a program is properly synchronized then it will exhibit sequentially consistent behavior. Finally, we use our model to prove that the unusual design choice of a partial order among writes to the same location is unobservable in any program.

1 INTRODUCTION

The original Java memory model [Manson et al. 2005] included an early attempt to define the semantics of lock-free shared memory programs running on the Java platform, but the definitions were hard to understand and there was no easy way to check the behavior of example programs. It was also later discovered that it ruled out existing compiler optimizations which it claimed to support [Ševčík and Aspinall 2008]. Since then, researchers have made great advances in memory model design while studying other weak memory models like those for ARM [Alglave et al. 2008; Pulte et al. 2017], C11 [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017; Vafeiadis et al. 2015], Power [Alglave et al. 2014], and x86 [Owens et al. 2009].

Recently, the ninth version of the Java Development Kit updated and expanded Java’s memory model using new "access modes". Though the design of the access modes is inspired by C11’s memory orders [Committee et al. 2010], it differs in a few key ways. First, it sheds complicated legacy features like release sequences and release-consume accesses. Second, it includes a broad but simple mechanism to forbid so called “out of thin-air” behavior [Batty and Sewell 2014]. Finally, it makes no provision for a total order on writes to the same location. Taken together this suggests new opportunities to use a simpler model, develop metatheory, and verify lock-free algorithms for the Java platform.

However, the documentation [JDK9 2017; Lea 2017, 2018] is frequently ambiguous. This makes it extremely difficult to provide definitive answers about program behavior and there is little hope of proving important properties about the semantics. Further, it impedes the discussion of key features of the model’s design.

To address these issues, we present the first formalization of Java’s access modes. Critically,

1. Alglave et al. 2014

Model

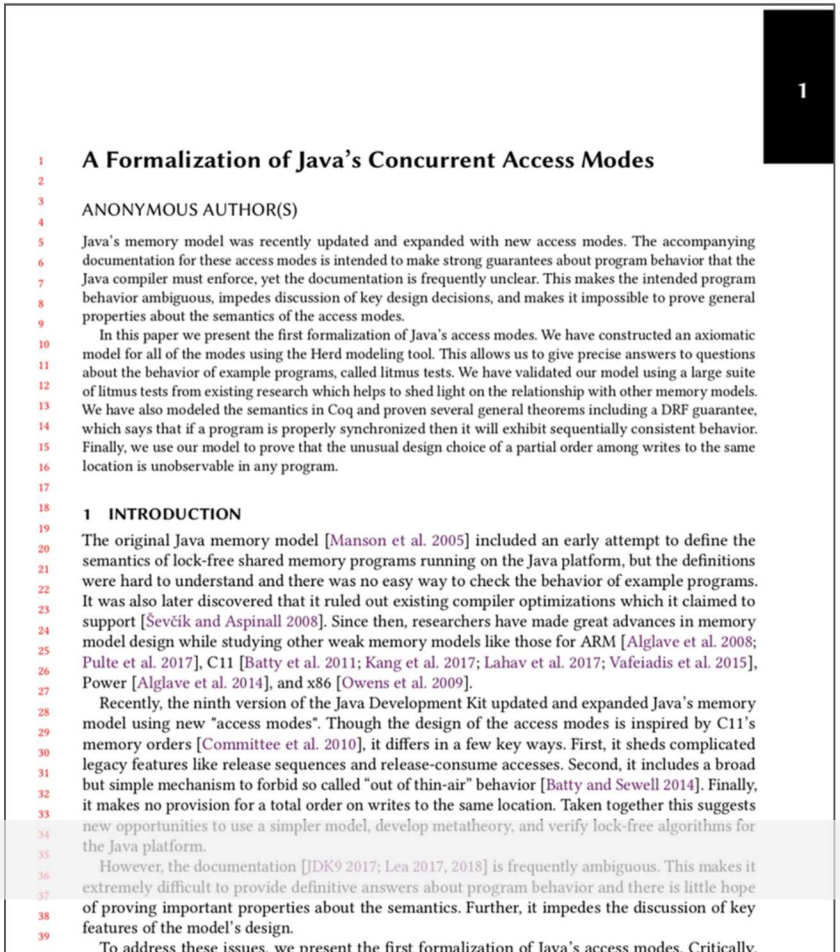
OOPSLA ‘19

Java Access Modes (JAM) Model

Instantiations in Herd[1] and Coq

Litmus Test Suite

80+ example programs



1

A Formalization of Java’s Concurrent Access Modes

ANONYMOUS AUTHOR(S)

Java’s memory model was recently updated and expanded with new access modes. The accompanying documentation for these access modes is intended to make strong guarantees about program behavior that the Java compiler must enforce, yet the documentation is frequently unclear. This makes the intended program behavior ambiguous, impedes discussion of key design decisions, and makes it impossible to prove general properties about the semantics of the access modes.

In this paper we present the first formalization of Java’s access modes. We have constructed an axiomatic model for all of the modes using the Herd modeling tool. This allows us to give precise answers to questions about the behavior of example programs, called litmus tests. We have validated our model using a large suite of litmus tests from existing research which helps to shed light on the relationship with other memory models. We have also modeled the semantics in Coq and proven several general theorems including a DRF guarantee, which says that if a program is properly synchronized then it will exhibit sequentially consistent behavior. Finally, we use our model to prove that the unusual design choice of a partial order among writes to the same location is unobservable in any program.

1 INTRODUCTION

The original Java memory model [Manson et al. 2005] included an early attempt to define the semantics of lock-free shared memory programs running on the Java platform, but the definitions were hard to understand and there was no easy way to check the behavior of example programs. It was also later discovered that it ruled out existing compiler optimizations which it claimed to support [Ševčík and Aspinall 2008]. Since then, researchers have made great advances in memory model design while studying other weak memory models like those for ARM [Alglave et al. 2008; Pulte et al. 2017], C11 [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017; Vafeiadis et al. 2015], Power [Alglave et al. 2014], and x86 [Owens et al. 2009].

Recently, the ninth version of the Java Development Kit updated and expanded Java’s memory model using new "access modes". Though the design of the access modes is inspired by C11’s memory orders [Committee et al. 2010], it differs in a few key ways. First, it sheds complicated legacy features like release sequences and release-consume accesses. Second, it includes a broad but simple mechanism to forbid so called “out of thin-air” behavior [Batty and Sewell 2014]. Finally, it makes no provision for a total order on writes to the same location. Taken together this suggests new opportunities to use a simpler model, develop metatheory, and verify lock-free algorithms for the Java platform.

However, the documentation [JDK9 2017; Lea 2017, 2018] is frequently ambiguous. This makes it extremely difficult to provide definitive answers about program behavior and there is little hope of proving important properties about the semantics. Further, it impedes the discussion of key features of the model’s design.

To address these issues, we present the first formalization of Java’s access modes. Critically,

1. Alglave et al. 2014

Model

OOPSLA ‘19

Java Access Modes (JAM) Model

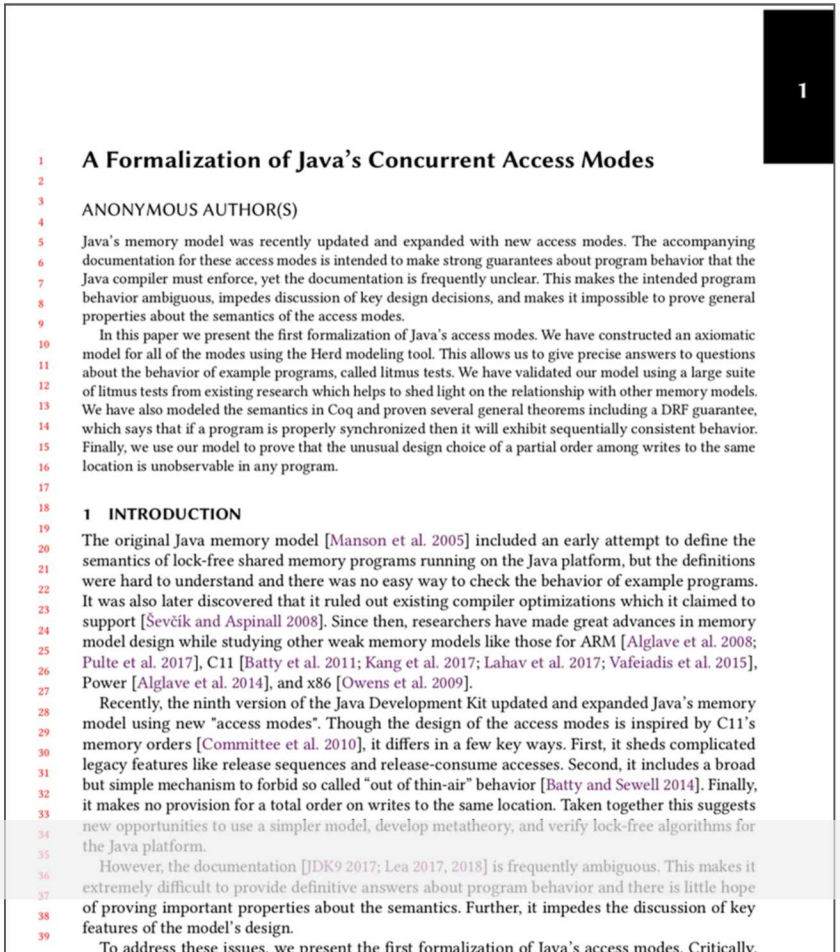
Instantiations in Herd[1] and Coq

Litmus Test Suite

80+ example programs

Theorems

3 main theorems, unobservability



1

A Formalization of Java’s Concurrent Access Modes

ANONYMOUS AUTHOR(S)

Java’s memory model was recently updated and expanded with new access modes. The accompanying documentation for these access modes is intended to make strong guarantees about program behavior that the Java compiler must enforce, yet the documentation is frequently unclear. This makes the intended program behavior ambiguous, impedes discussion of key design decisions, and makes it impossible to prove general properties about the semantics of the access modes.

In this paper we present the first formalization of Java’s access modes. We have constructed an axiomatic model for all of the modes using the Herd modeling tool. This allows us to give precise answers to questions about the behavior of example programs, called litmus tests. We have validated our model using a large suite of litmus tests from existing research which helps to shed light on the relationship with other memory models. We have also modeled the semantics in Coq and proven several general theorems including a DRF guarantee, which says that if a program is properly synchronized then it will exhibit sequentially consistent behavior. Finally, we use our model to prove that the unusual design choice of a partial order among writes to the same location is unobservable in any program.

1 INTRODUCTION

The original Java memory model [Manson et al. 2005] included an early attempt to define the semantics of lock-free shared memory programs running on the Java platform, but the definitions were hard to understand and there was no easy way to check the behavior of example programs. It was also later discovered that it ruled out existing compiler optimizations which it claimed to support [Ševčík and Aspinall 2008]. Since then, researchers have made great advances in memory model design while studying other weak memory models like those for ARM [Alglave et al. 2008; Pulte et al. 2017], C11 [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2017; Vafeiadis et al. 2015], Power [Alglave et al. 2014], and x86 [Owens et al. 2009].

Recently, the ninth version of the Java Development Kit updated and expanded Java’s memory model using new "access modes". Though the design of the access modes is inspired by C11’s memory orders [Committee et al. 2010], it differs in a few key ways. First, it sheds complicated legacy features like release sequences and release-consume accesses. Second, it includes a broad but simple mechanism to forbid so called “out of thin-air” behavior [Batty and Sewell 2014]. Finally, it makes no provision for a total order on writes to the same location. Taken together this suggests new opportunities to use a simpler model, develop metatheory, and verify lock-free algorithms for the Java platform.

However, the documentation [JDK9 2017; Lea 2017, 2018] is frequently ambiguous. This makes it extremely difficult to provide definitive answers about program behavior and there is little hope of proving important properties about the semantics. Further, it impedes the discussion of key features of the model’s design.

To address these issues, we present the first formalization of Java’s access modes. Critically,

1. Alglave et al. 2014

Validation

Validation

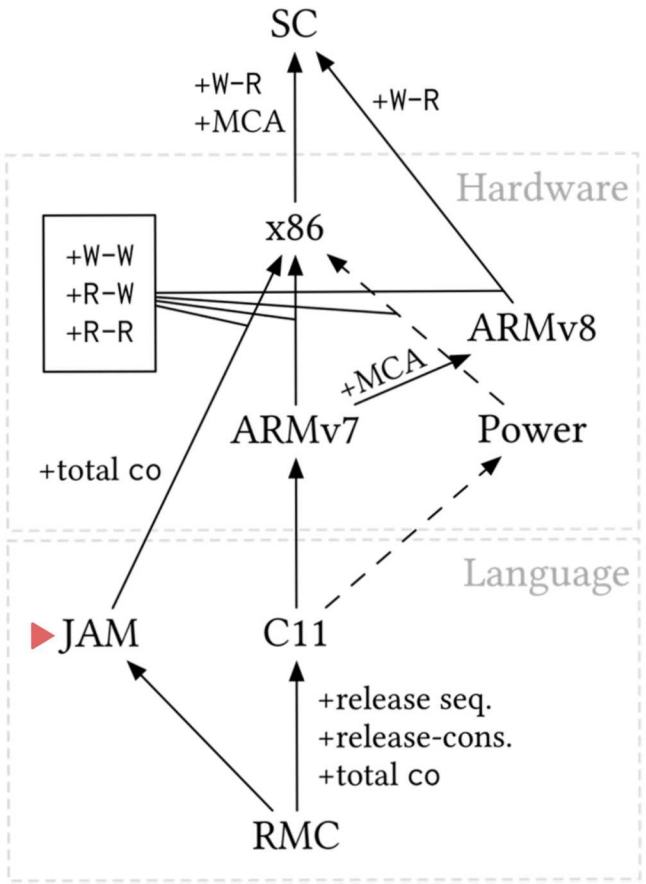
Empirical Validation

Testing specified expectations of the JAM

Validation

Empirical Validation

Testing specified expectations of the JAM



Validation

Empirical Validation

Testing specified expectations of the JAM

Theoretical Validation

Metatheorems required by the specification

OOPSLA'19

Validation

Empirical Validation

Testing specified expectations of the JAM

Theoretical Validation

Metatheorems required by the specification

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.

Theorem monotonicity { H1 H2 } :
  acyclic (co H1)
  -> trco H2
  -> match H2 H1
  -> ~ fiat_vo H2
  -> access_lte_ordered H2 H1
  -> acyclic (co H2).
Proof.
  …
Qed.
```

Validation

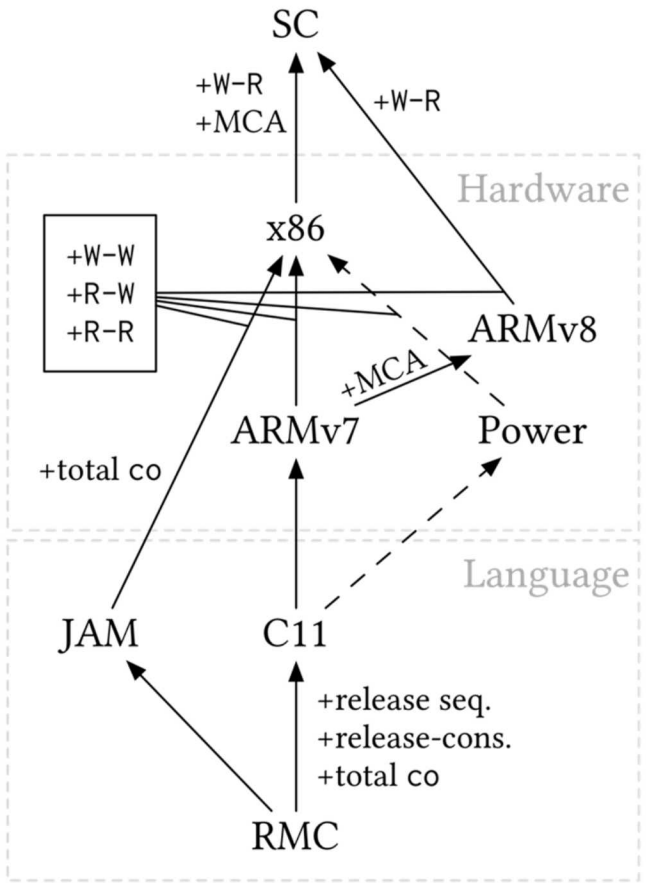
JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Validation

JAM vs. x86, ARMv8, C11

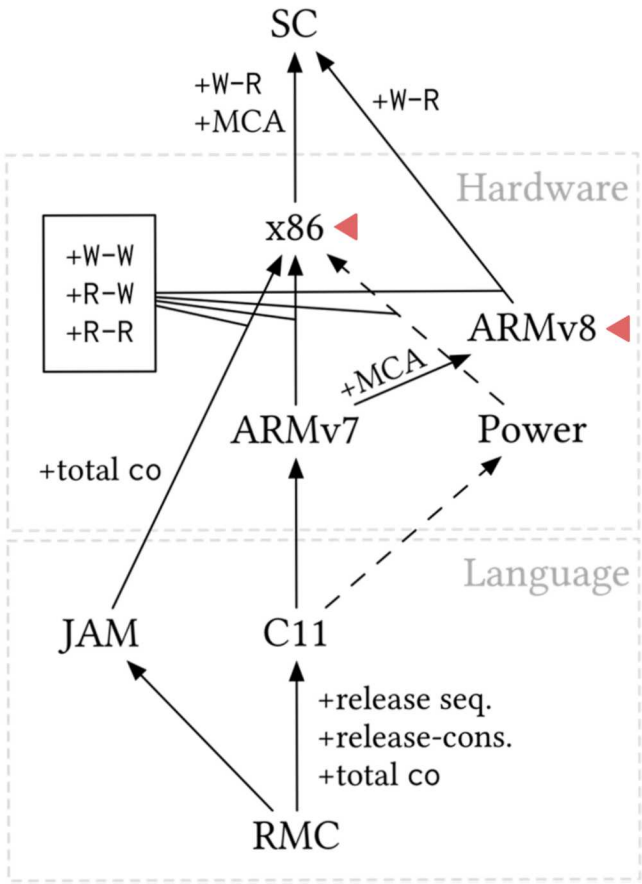
JAM should admit more behaviors/executions



Validation

JAM vs. x86, ARMv8, C11

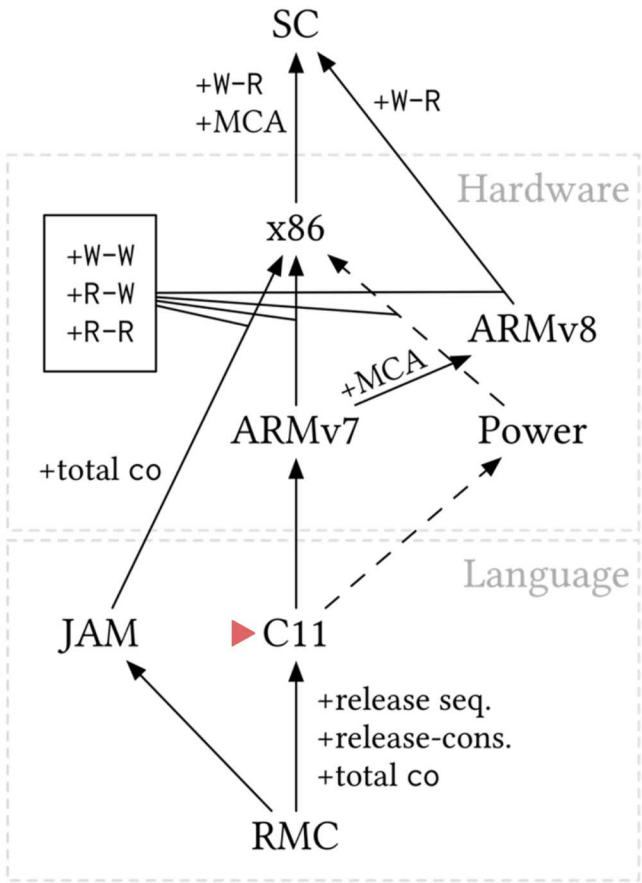
JAM should admit more behaviors/executions



Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions



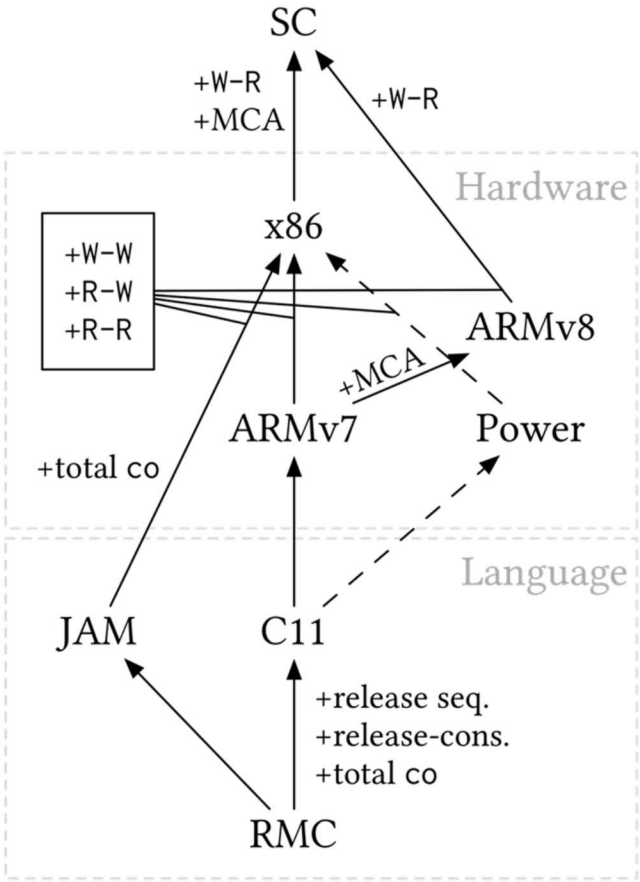
Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate



1. Alglave et al. 2014

Validation

JAM vs. x86, ARMv8, C11

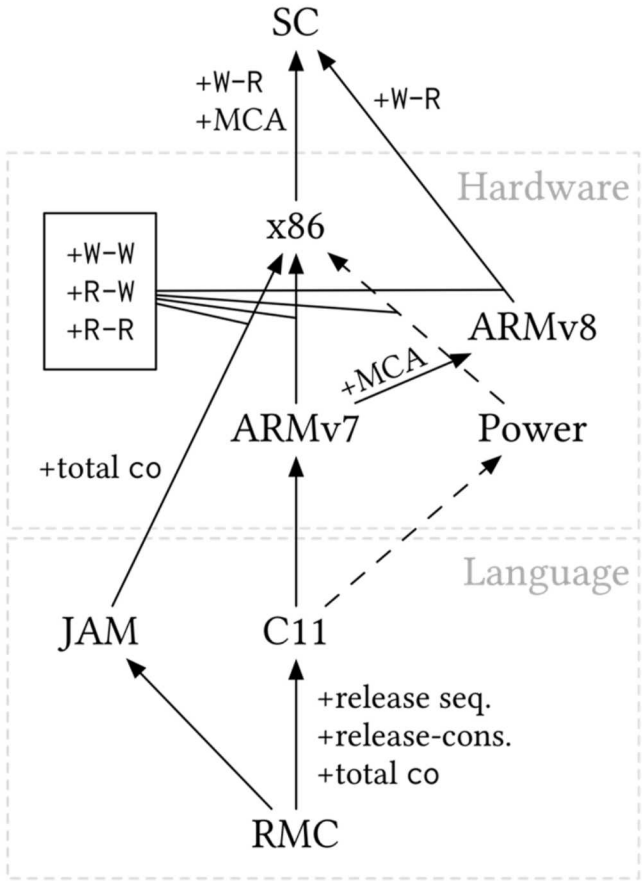
JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate

80+ Litmus Tests

Describe “behaviors”



1. Alglave et al. 2014

Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate

80+ Litmus Tests

Describe “behaviors”

```
thread0:        thread1:
y = 1;          a = x;
x = 1;          b = y;
```

1. Alglave et al. 2014

Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate

80+ Litmus Tests

Describe “behaviors”

1. Alglave et al. 2014

Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate

80+ Litmus Tests

Describe “behaviors”

1. Alglave et al. 2014

Validation

JAM vs. x86, ARMv8, C11

JAM should admit more behaviors/executions

Herd[1] Tool

Explore executions, consult model to validate

80+ Litmus Tests

Describe “behaviors”

SC ✗ ARMv8 ✓

1. Alglave et al. 2014

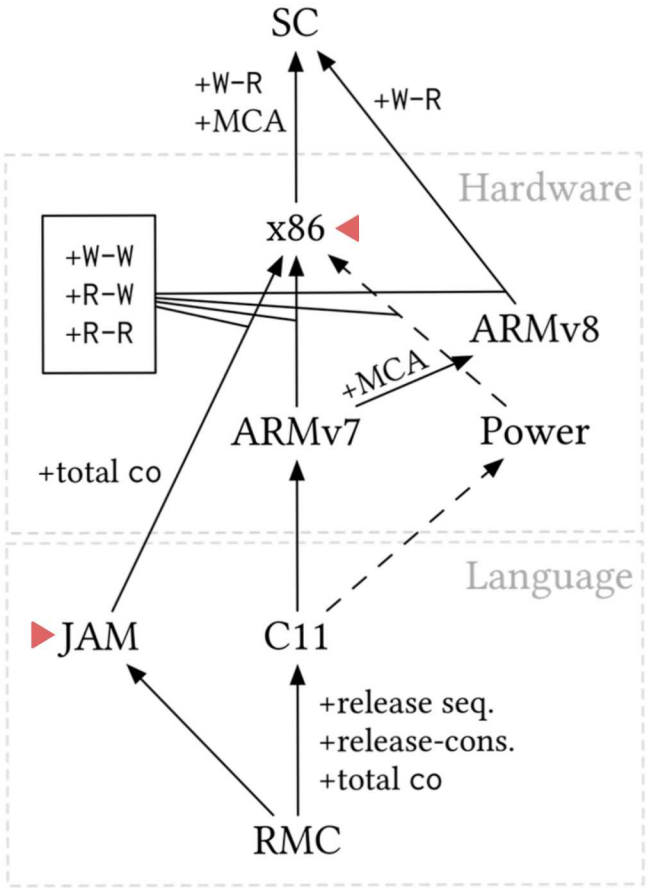
JAM vs. x86

SC
¬W R
MCA
W-R
x86
¬W-W
¬R-W
¬R-R
ARMv8
¬MCA
ARMv7
Power
total co
JAM
C11
release seq.
release cons.
total co
RAIC

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions



JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. x86

Expectation: the JAM is weaker

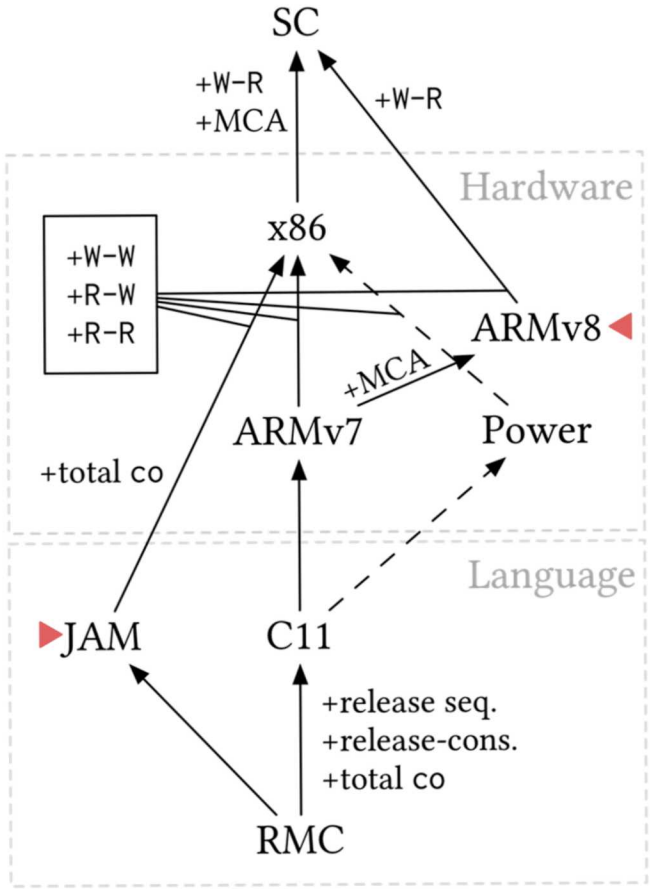
Permits more behaviors/executions

Allowed vs. Not Allowed

The behavior is allowed if one execution exists

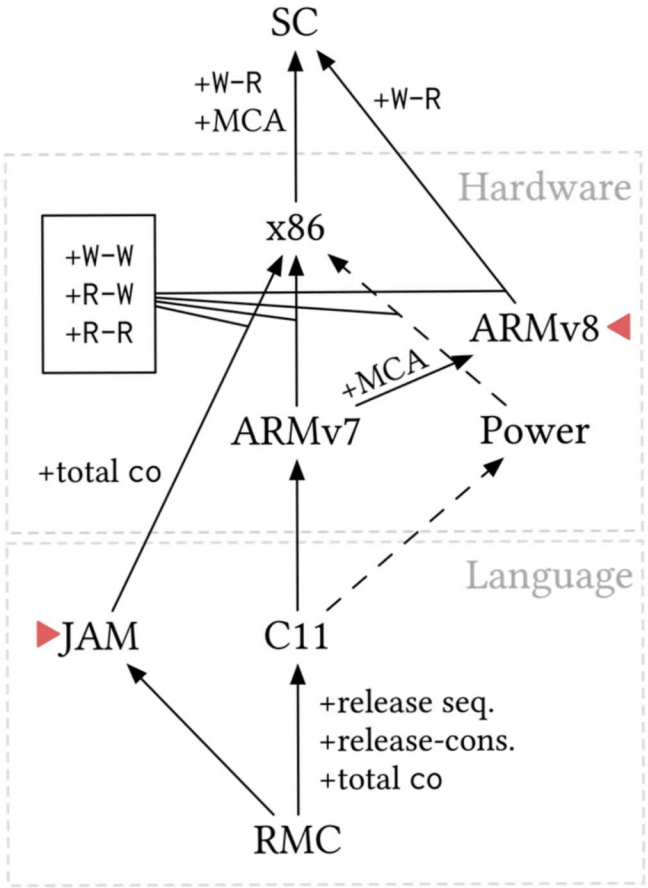
x86	JAM	Count
Allowed	Allowed	16
Not Allowed	Allowed	2
Not Allowed	Not Allowed	2
Allowed	Not Allowed	0

JAM vs. ARMv8


SC
+W-R
+MCA
+W-R
Hardware
x86
+W-W
+R-W
+R-R
ARMv8
+MCA
ARMv7
Power
+total co
Language
JAM
C11
+release seq.
+release-cons.
+total co
RMC

JAM vs. ARMv8

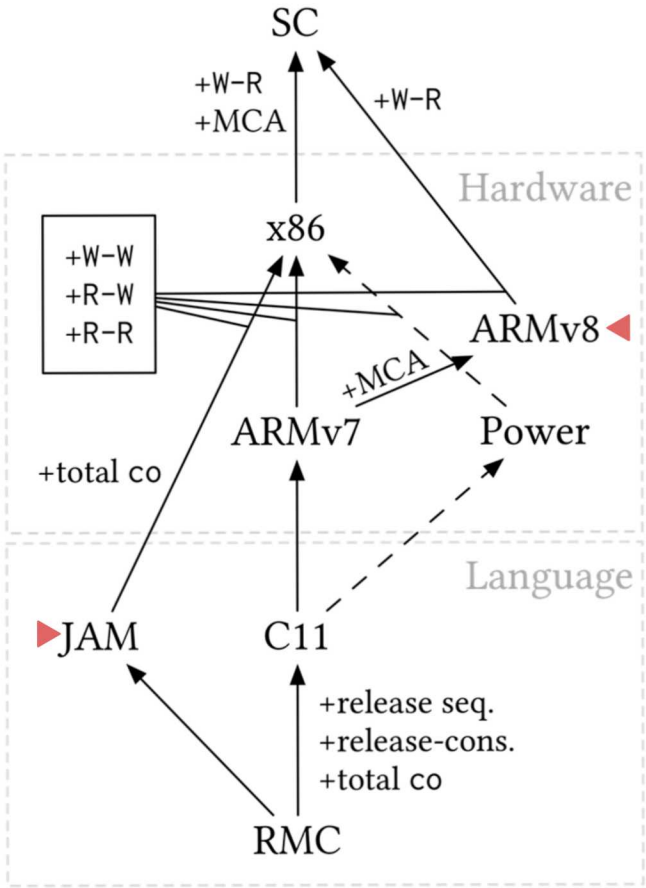
Expectation: the JAM is weaker



JAM vs. ARMv8

Expectation: the JAM is weaker

Except for causal cycles



JAM vs. ARMv8

Expectation: the JAM is weaker

Except for causal cycles

ARMv8	JAM	Count
Allowed	Allowed	2
Not Allowed	Allowed	5
Not Allowed	Not Allowed	4
Allowed	Not Allowed	1

JAM vs. ARMv8

Expectation: the JAM is weaker

Except for causal cycles

ARMv8	JAM	Count
Allowed	Allowed	2
Not Allowed	Allowed	5
Not Allowed	Not Allowed	4
Allowed	Not Allowed	1

JAM vs. ARMv8

Expectation: the JAM is weaker

Except for causal cycles

ARMv8	JAM	Count
Allowed	Allowed	2
Not Allowed	Allowed	5
Not Allowed	Not Allowed	4
Allowed	Not Allowed	1

JAM vs. ARMv8

Expectation: the JAM is weaker

Except for causal cycles

Load Buffering (LB), Causal Cycles

Forbidden explicitly in the JAM

ARMv8	JAM	Count
Allowed	Allowed	2
Not Allowed	Allowed	5
Not Allowed	Not Allowed	4
Allowed	Not Allowed	1

JAM vs. ARMv8

Expectation: the JAM is weaker

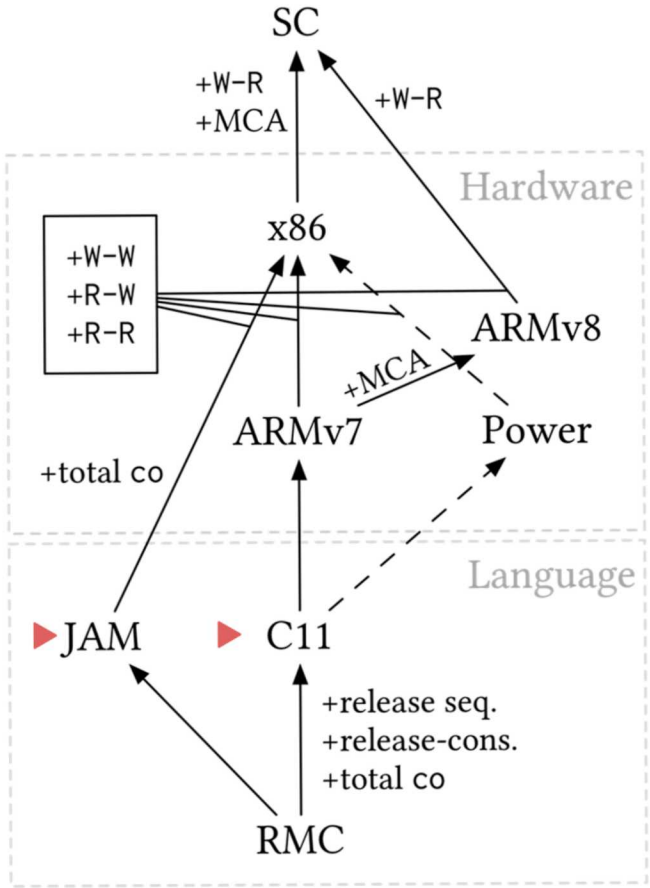
Except for causal cycles

Load Buffering (LB), Causal Cycles

Forbidden explicitly in the JAM

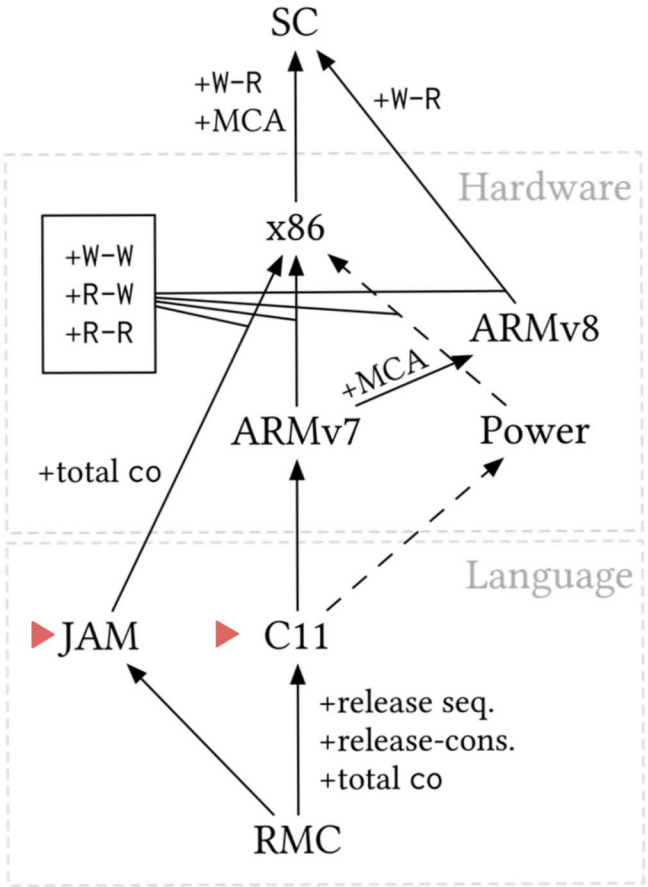
ARMv8	JAM	Count
Allowed	Allowed	2
Not Allowed	Allowed	5
Not Allowed	Not Allowed	4
Allowed	Not Allowed	1

JAM vs. C11


SC
+W-R
+MCA
+W-R
Hardware
x86
+W-W
+R-W
+R-R
ARMv8
+MCA
ARMv7
Power
+total co
Language
JAM
C11
+release seq.
+release-cons.
+total co
RMC

JAM vs. C11

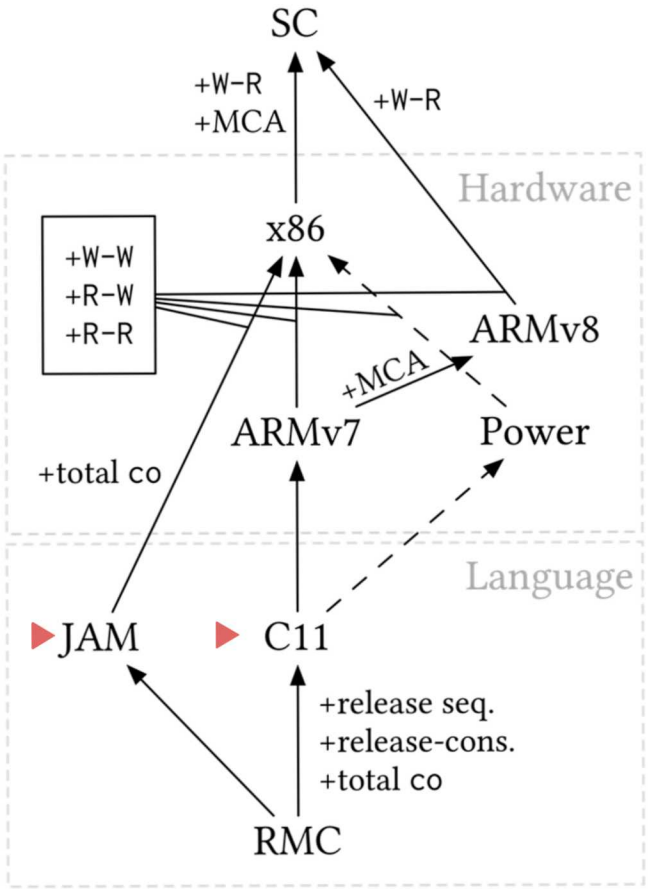
Expectation: the JAM is weaker



JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles



JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles

C11	JAM	Count
Allowed	Allowed	12
Not Allowed	Allowed	2
Not Allowed	Not Allowed	33
Allowed	Not Allowed	1

JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles

C11	JAM	Count
Allowed	Allowed	12
Not Allowed	Allowed	2
Not Allowed	Not Allowed	33
Allowed	Not Allowed	1

JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles

C11	JAM	Count
Allowed	Allowed	12
Not Allowed	Allowed	2
Not Allowed	Not Allowed	33
Allowed	Not Allowed	1

JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles

Load Buffering (LB), Causal Cycles

Forbidden explicitly in the JAM

C11	JAM	Count
Allowed	Allowed	12
Not Allowed	Allowed	2
Not Allowed	Not Allowed	33
Allowed	Not Allowed	1

JAM vs. C11

Expectation: the JAM is weaker

Except for causal cycles

Load Buffering (LB), Causal Cycles

Forbidden explicitly in the JAM

C11	JAM	Count
Allowed	Allowed	12
Not Allowed	Allowed	2
Not Allowed	Not Allowed	33
Allowed	Not Allowed	1

Theorems

Theorems

Three main theorems

Further validation of our semantics

1. Ou and Demsky 2018

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.
```

1. Ou and Demsky 2018

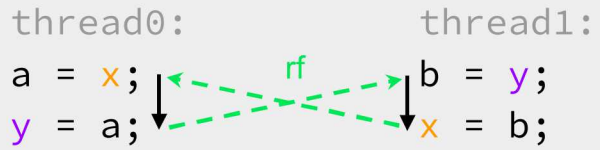
Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.
```



1. Ou and Demsky 2018

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.
```

1. Ou and Demsky 2018

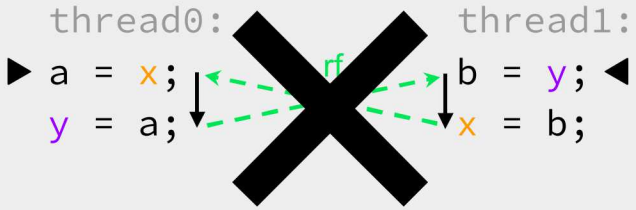
Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.
```



1. Ou and Demsky 2018

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.
```

1. Ou and Demsky 2018

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]
- SC semantics with proper synchronization

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.
```

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]
- SC semantics with proper synchronization
- Monotonicity of access modes

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.

Theorem monotonicity { H1 H2 } :
  acyclic (co H1)
  -> trco H2
  -> match H2 H1
  -> ~ fiat_vo H2
  -> access_lte_ordered H2 H1
  -> acyclic (co H2).
Proof.
  …
Qed.
```

Theorems

Three main theorems

Further validation of our semantics

- Forbidding causal cycles w/ acquire reads [1]
- SC semantics with proper synchronization
- Monotonicity of access modes

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.

Theorem monotonicity { H1 H2 } :
  acyclic (co H1)
  -> acyclic (co H2).
Proof.
  …
Qed.
```

plain ⊑ opaque ⊑ release-acquire ⊑ volatile

OOPSLA'19

Theorems

Three main theorems

Further validation of our semantics

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.

Theorem monotonicity { H1 H2 } :
  acyclic (co H1)
  -> trco H2
  -> match H2 H1
  -> ~ fiat_vo H2
  -> access_lte_ordered H2 H1
  -> acyclic (co H2).
Proof.
  …
Qed.
```

Theorems

Three main theorems

Further validation of our semantics

Unobservable partial co

Impossible to construct a litmus test

```
Theorem acq_causality { H } :
  trco H
  -> acquire_reads H
  -> opaque_accesses H
  -> acyclic (union (po H) (rf H)).
Proof.
  …
Qed.

Theorem drf_sc { H } :
  trco H
  -> race_free H
  -> acyclic (co H)
  -> acyclic (sc (po H) H).
Proof.
  …
Qed.

Theorem monotonicity { H1 H2 } :
  acyclic (co H1)
  -> trco H2
  -> match H2 H1
  -> ~ fiat_vo H2
  -> access_lte_ordered H2 H1
  -> acyclic (co H2).
Proof.
  …
Qed.
```

Future Work

Update Java language spec
Documentation, Java stress tests

Cost of forbidding causal cycles
Performance evaluation, optimization techniques

Fuzzing Hotspot
Find behaviors allowed by VM, not by model

Logic for specified orders
Reasoning in SC, proof in Opaque Mode

Specified orders for crash protocols
Replace fsync with specified orders

Specified Orders as hardware synch.
Evaluating performance benefits over fences

Unified semantics for DS and Java
Partial order on writes is DS-like

Thanks!

questions?