What is a Memory Model?

Jaeheon Yi

October 22, 2008

TWIGS

That code doesn't work?

```
red == blue == 0
<p1>                    <p2>
red = 1;                blue = 1;
if(blue==0)             if(red==0)
  //critical              //critical
  //section               //section
```

Architectural Trends

- Relative cost of memory access expensive
  - 100's of cycles to access
- Processor speed difficult to increase
  - Ridiculous energy consumption

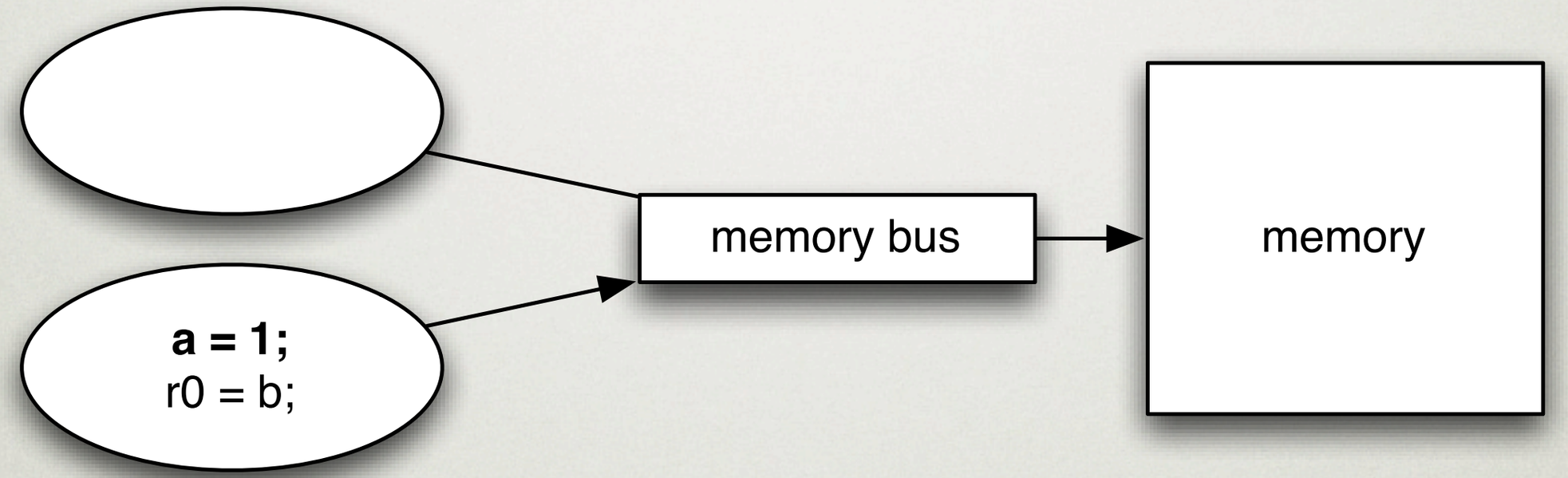
Implications for Software

- Everybody has multicore, soon manycore
- Dominance of shared-memory paradigm
- Multiple threads of execution for performance
- H/W and compilers try to hide memory latencies

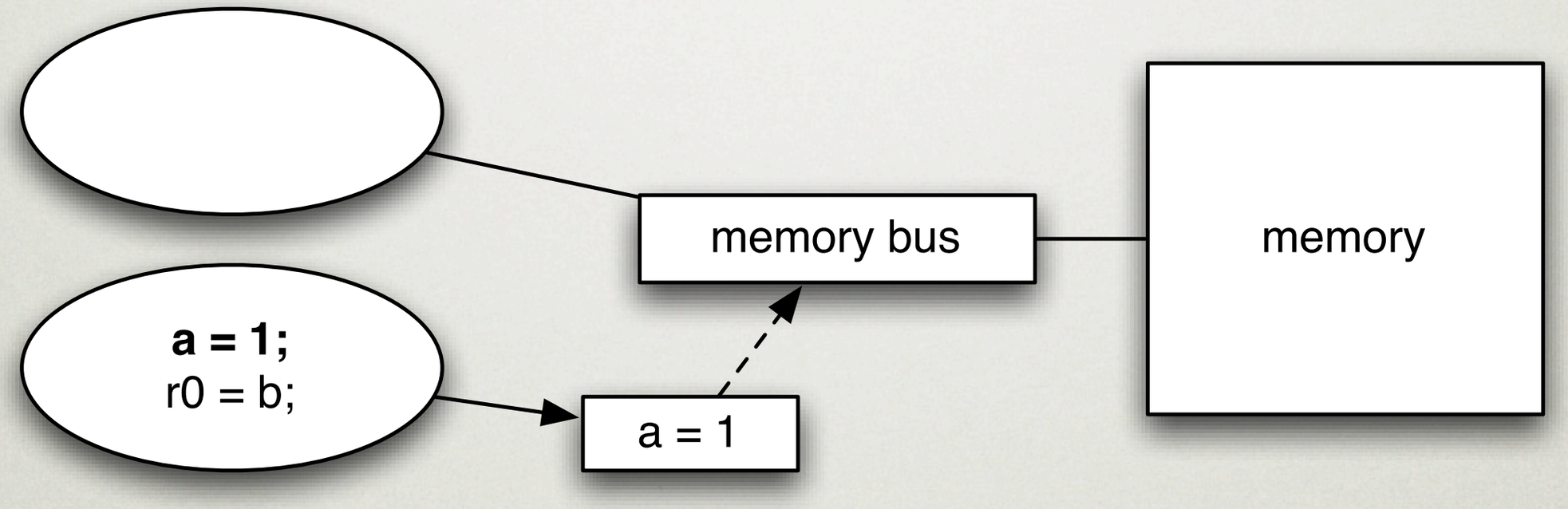
Hardware Funny Business

- Some examples:
  - Write buffers
  - Caches at multiple levels
  - Out-of-order execution
  - Speculative execution

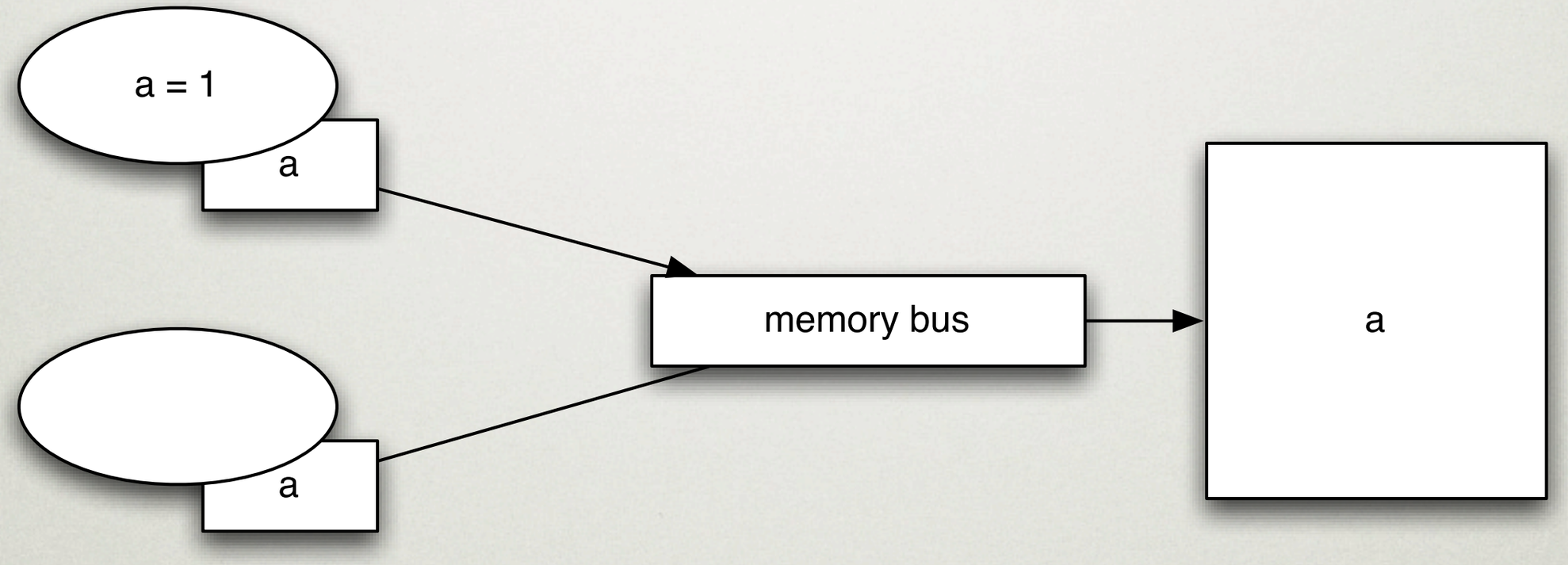
Write Buffers



Write Buffers



Caches



Out-of-Order Execution

```
r0 = x          ->      r0 = x
x = 3                   r1 = 2
r1 = 2                  x = 3
```

Speculative Execution

```
if(x==0)      ->      Assume x is 0
   y = 0                y = 0
r1 = 1                 Is x really 0? (y)
                       r1 = 1
```

Compiler Funny Business

- Some examples:
  - Loop-invariant code motion
  - Register allocation
  - Common sub-expression elimination

Loop-Inv. Code Motion

```
while (i < 100)
   a[i] = (b*3+3) + i
```

becomes

```
int r0 = b*3+3
while (i < 100)
   a[i] = r0 + i
```

Register Allocation

```
a = 3 * a          ->      r0 = a
c = b + 4                  r0 = r0 * 3
                           a = r0
                           r1 = b
                           r1 = r1 + 4
                           c = r1
```

Common Subexpression Elimination

```
x = 3*4 + a[i]*a[j]
y = a[i]*a[j] - a[k]
```

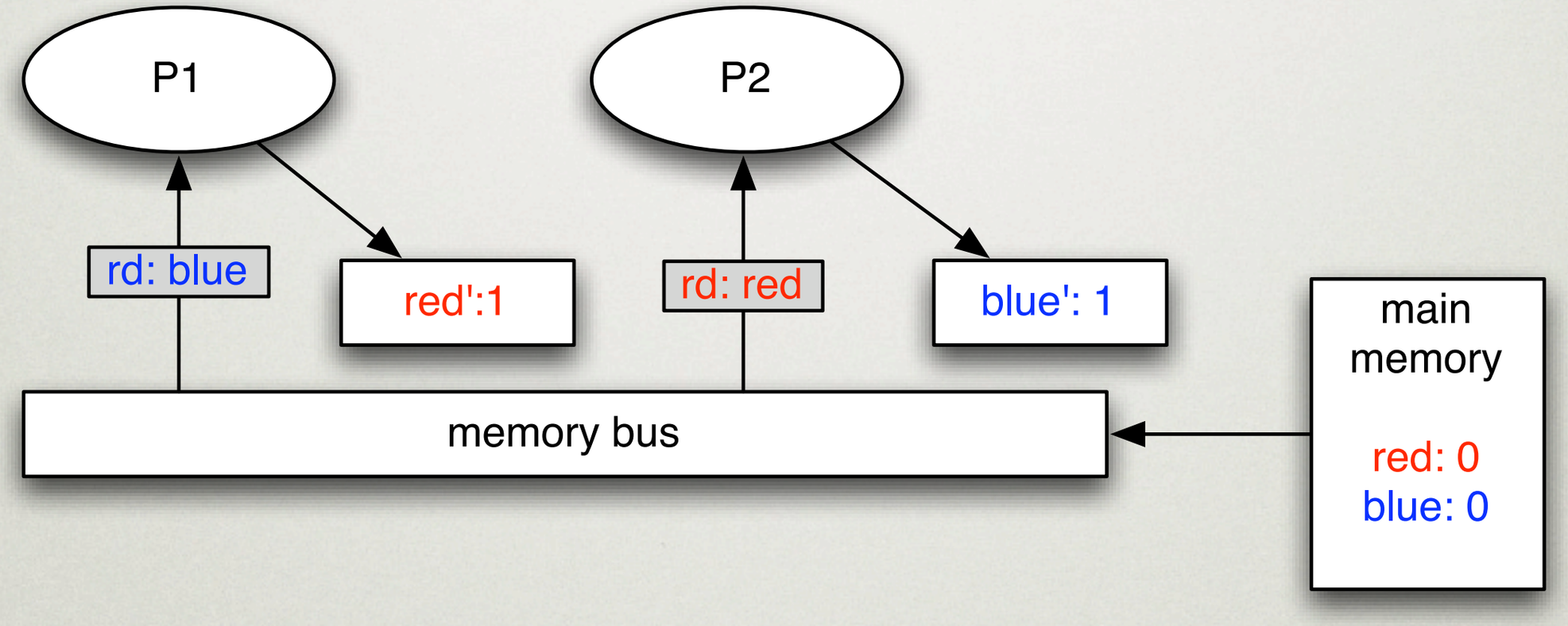
becomes

```
tmp = a[i]*a[j]
x = 3*4 + tmp
y = tmp - a[k]
```

That code doesn't work.

```
red == blue == 0
<p1>                    <p2>
red = 1;                blue = 1;
if(blue==0)             if(red==0)
  //critical              //critical
  //section               //section
```

That code doesn't work.



That code doesn't work.

```
red == blue == 0
<p1>                    <p2>
r0 = blue;              r0 = red;
red = 1;                blue = 1;
if(r0==0)               if(r0==0)
  //critical              //critical
  //section               //section
```

Memory Consistency Model

- Defines observable values
- Observation means “reads”
- What can the value be that you read out of memory?

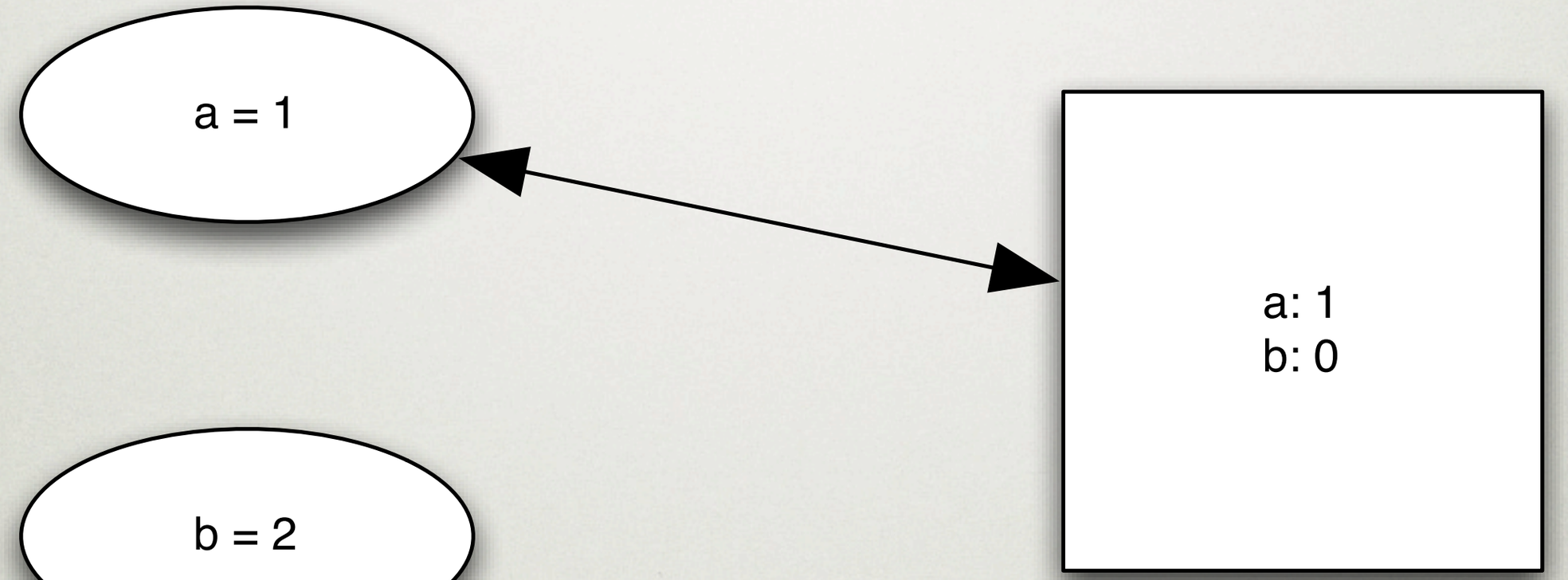
Memory Consistency Model

- Architecture level
  - x86 family has a memory model
- Language level
  - Java has one, C doesn't
  - C++ memory model in the works

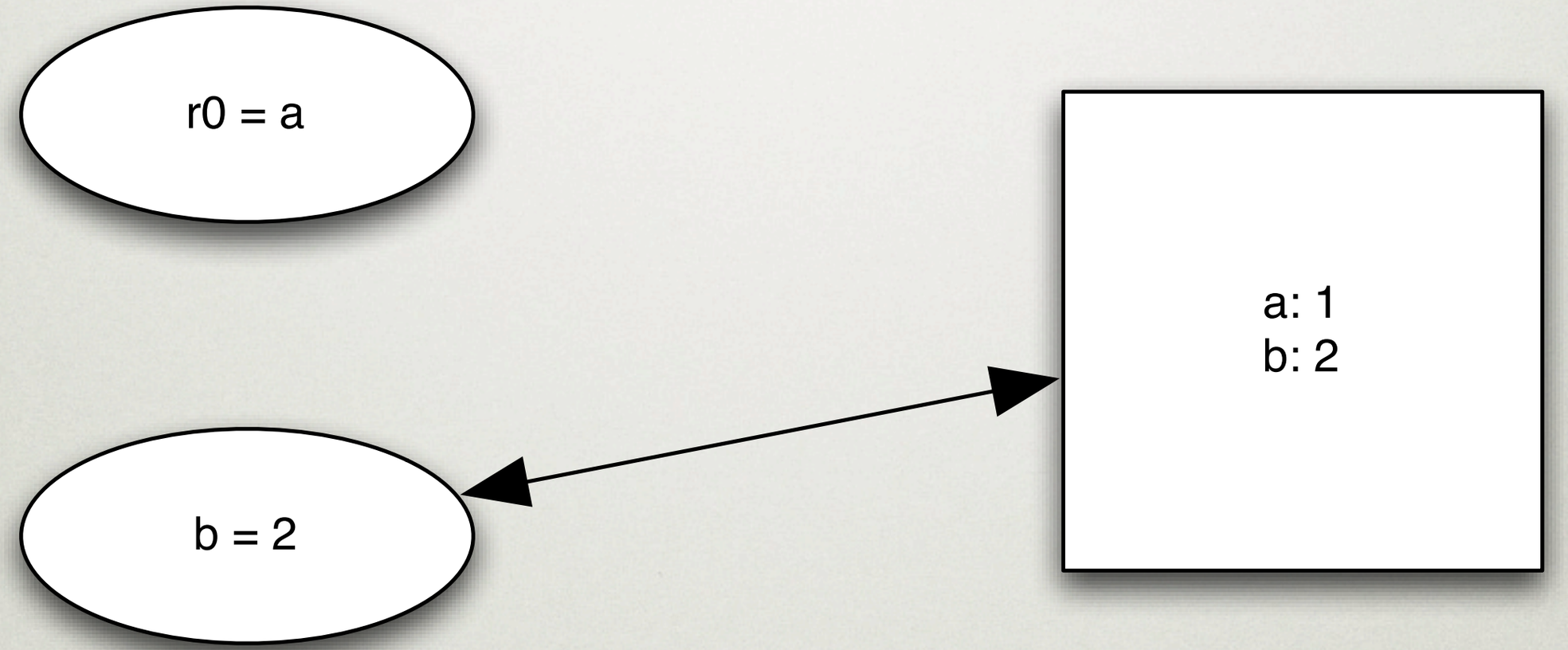
Sequential Consistency

- In each processor, instructions execute in program order
- Memory operations appear to execute one-at-a-time across entire system

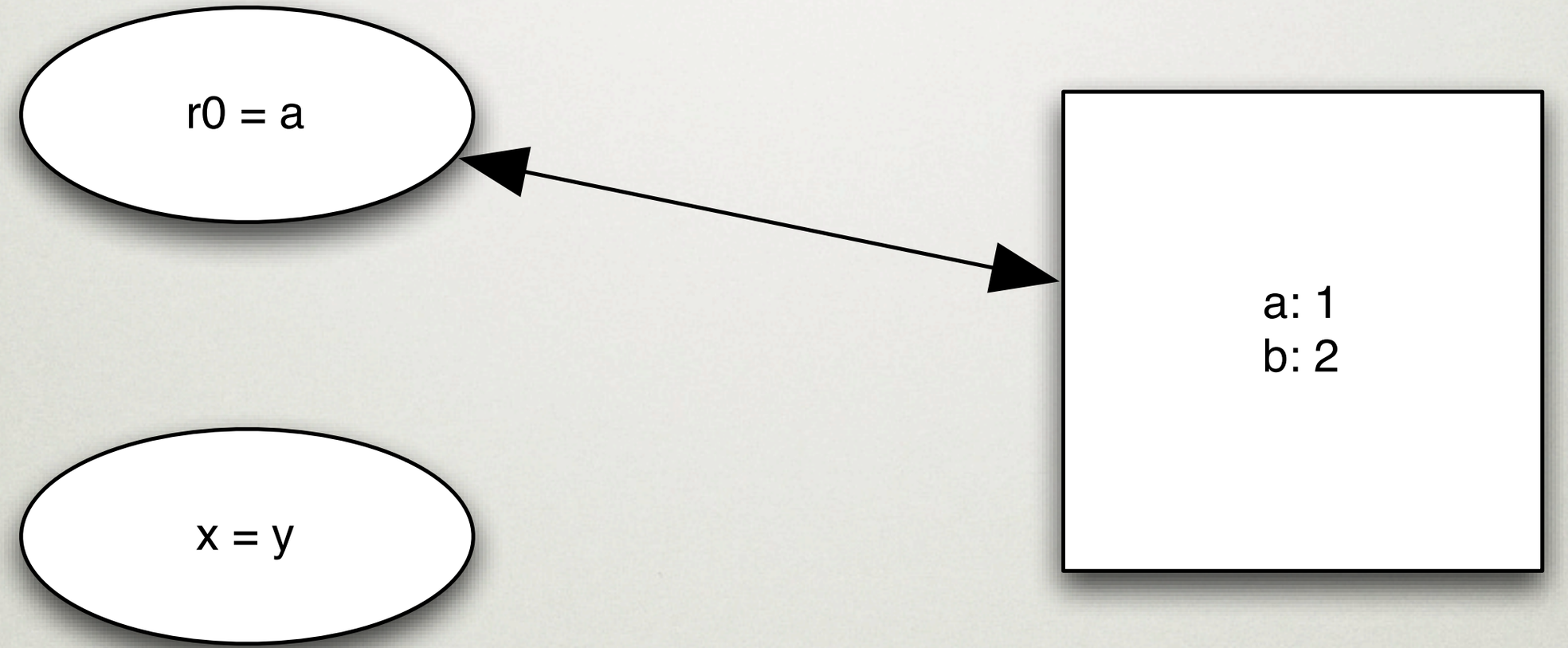
Sequential Consistency



Sequential Consistency



Sequential Consistency



Sequential Consistency

- Canonical, *intuitive,* most restrictive
- Prevents optimizations
  - Kills performance

Relaxed Memory Models

- Speeds up processors
- Harder to understand and program
- Many flavors

Ways to Relax

- Program order may be relaxed
- Write atomicity may be relaxed

Ways to Relax

- Program ordering requirement
  - Write(x); Read(y)
  - Write(x); Write(y)
  - Read(x); Read(y) / Write(y)

Ways to Relax

- Write atomicity requirement
  - $Write_1(x)$; early $Read_2(x)$
  - $Write_1(x)$; early $Read_1(x)$

Relaxing Too Much? Restoring Order

- Relaxed memory models often come with special instructions in ISA
  - Hardware synchronization primitives
- Expensive to use
- Fence, CAS, LL/SC

Relaxing Too Much? Restoring Order

- Memory fence instruction
  - Also called a memory barrier
  - Makes all pending writes visible to issuer
  - May cost 100's of cycles to execute
  - Load fences, store fences, full fences

Relaxing Too Much? Restoring Order

- Compare-And-Swap instruction
- Universal concurrency primitive
- Atomically executed write
- `bool CAS(addr, oldVal, newVal)`

Relaxing Too Much? Restoring Order

- Load-Linked/Store-Conditional instruction pair
- Universal concurrency primitive
- LL reads from address x
- SC attempts write to address x

Relaxing Too Much? Restoring Order

- Synchronization primitives to build
  - Mutex
  - Semaphore
  - Condition Variables
  - Monitors

Easier Way Out

- Reasoning about relaxations is not relaxing!
- Virtually all explicitly defined memory models have a safety hatch built in

Easier Way Out

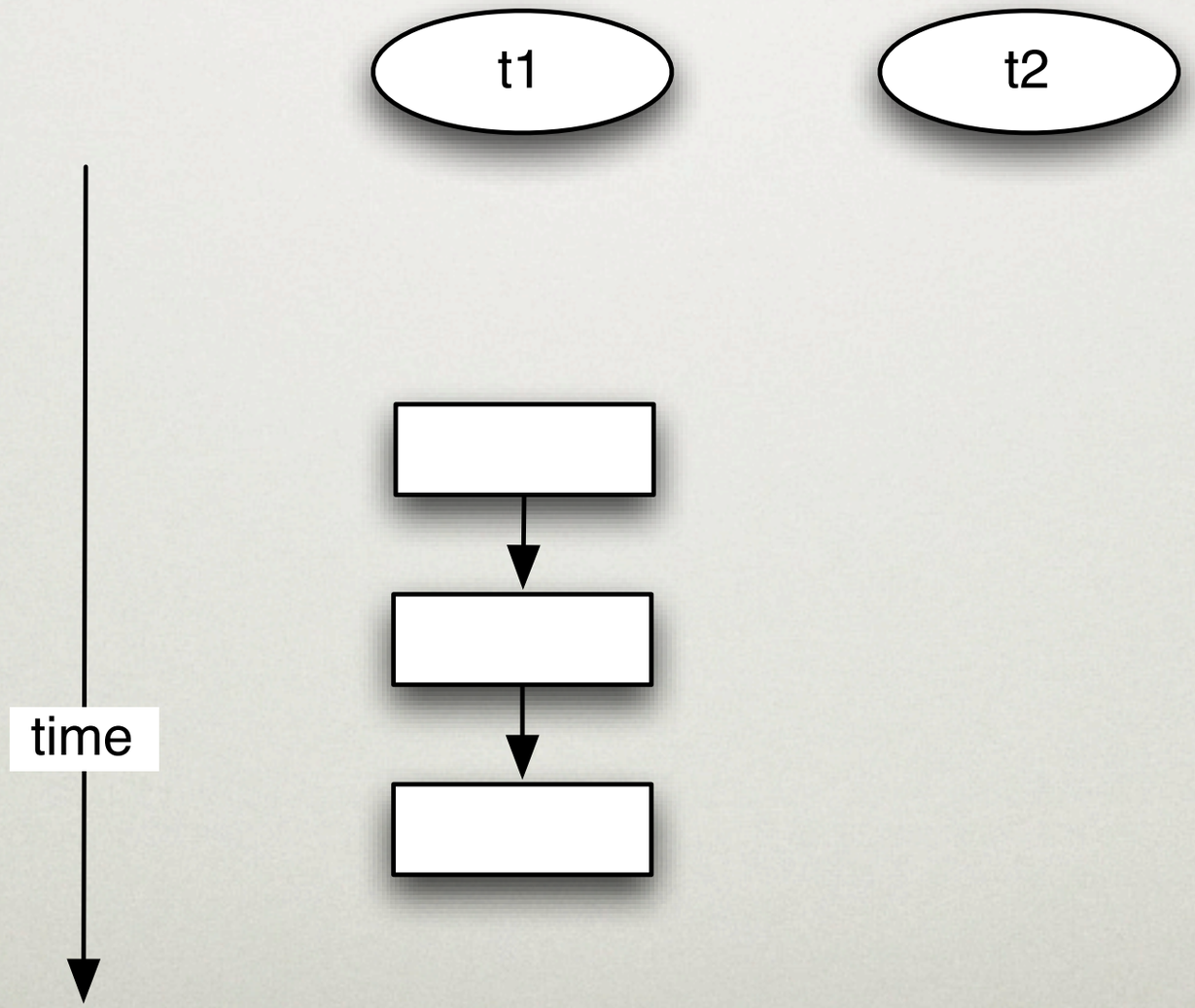
- Fundamental Property of Memory Models:
  - If your program is free of data races, then it will have sequentially consistent semantics

Easier Way Out

- Happens-before relation
  - A binary relation on executed instructions, transitively-closed
  - If x occurs before y in time, and x *conflicts with* y, then x *happens-before* y

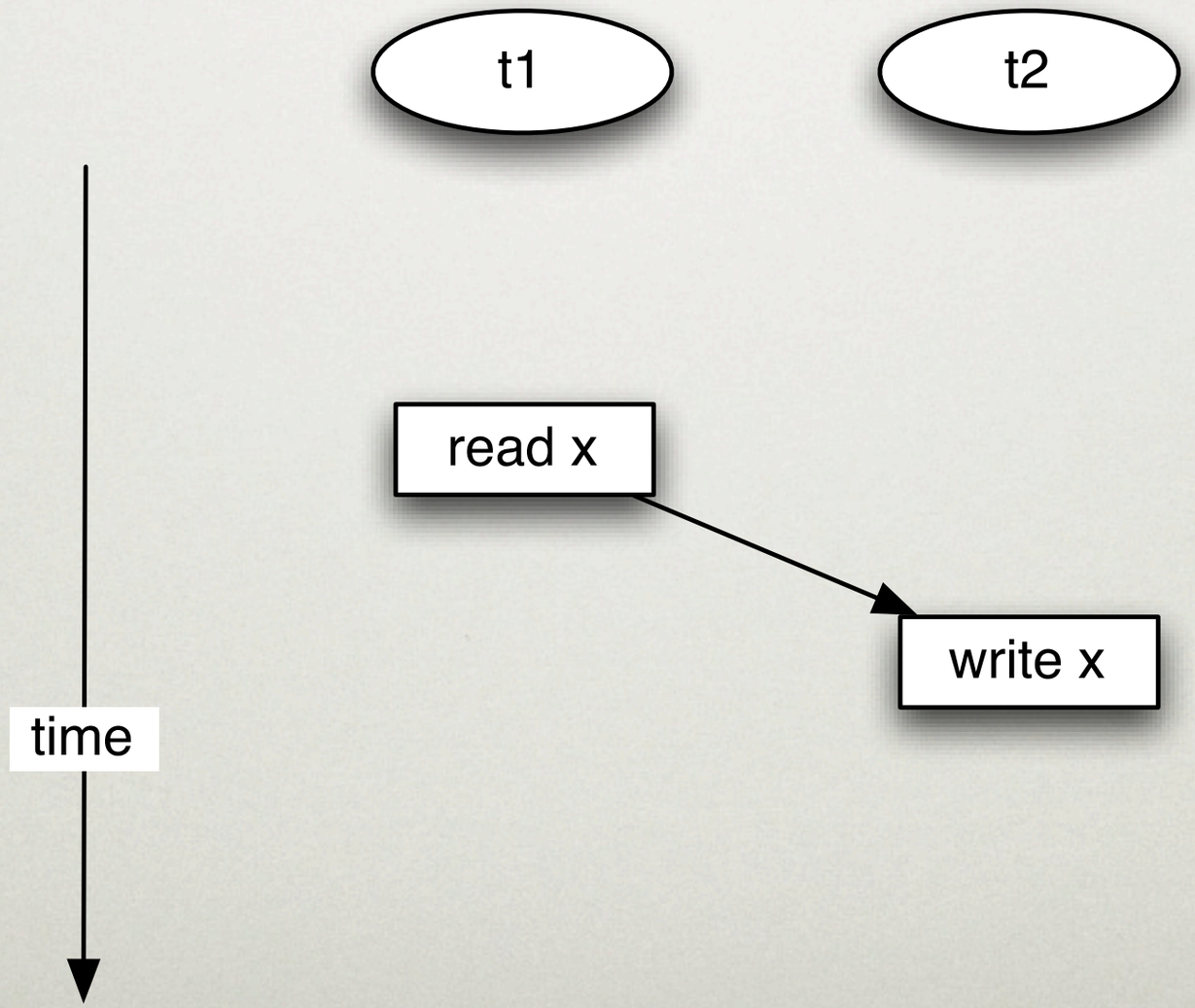
Easier Way Out

- x *conflicts with* y, program order



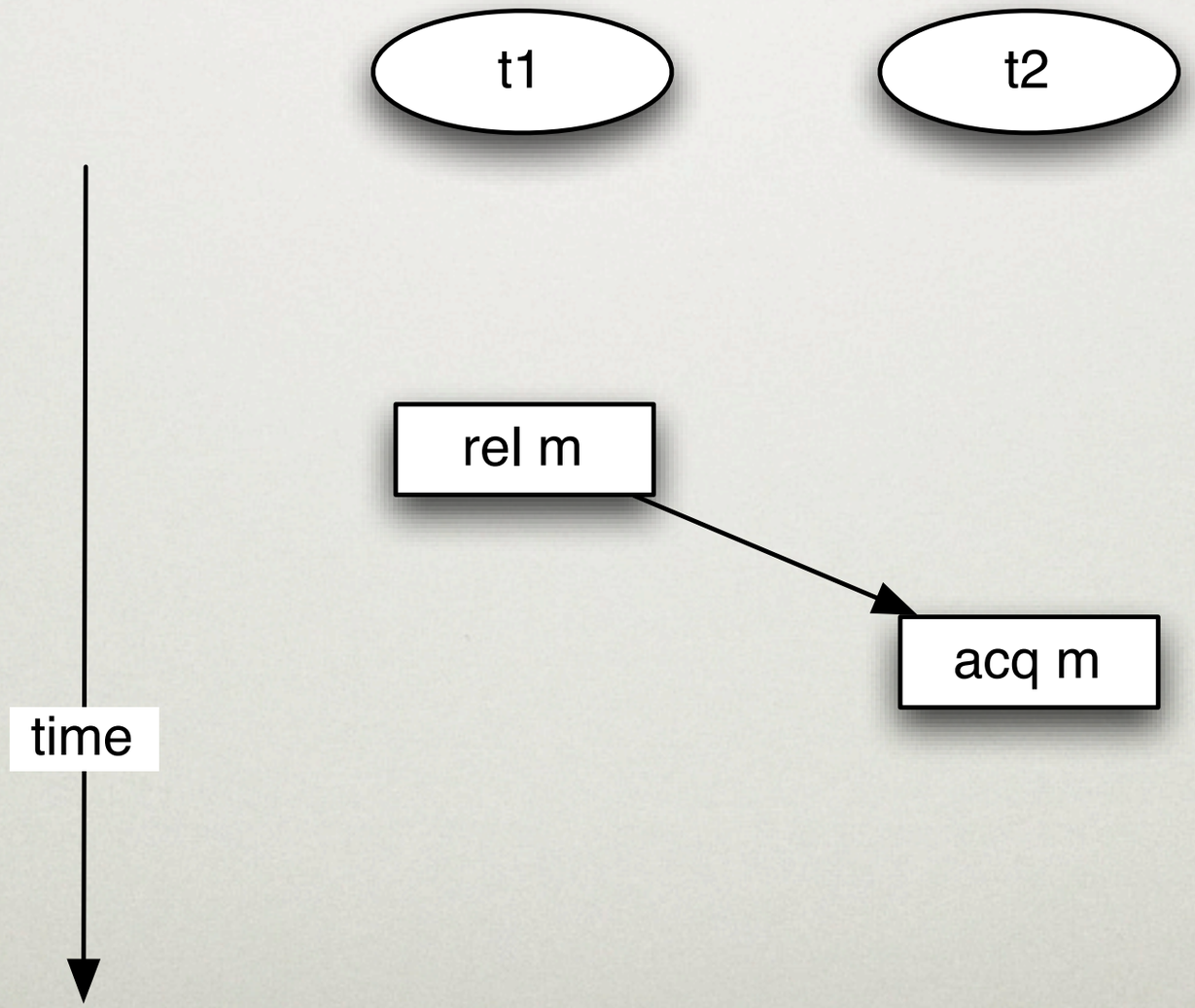
Easier Way Out

- x *conflicts with* y, communication order



Easier Way Out

- x *conflicts with* y, synchronization order



Easier Way Out

- Data race occurs when:
  - A variable is read by >1 threads
  - It is written by at least one thread
  - Read/write *not* ordered by synchronization edge

Easier Way Out

- Data-Race-Freedom (DRF) property: The absence of any data races in a program
- Defines synchronization property at the program level, not the execution trace level

x86 Memory Model

- Formally defined as of last year
- 8 principles define possible instruction orderings

x86 Memory Model

1. Loads are *not* reordered with other loads.
2. Stores are *not* reordered with other stores.
3. Stores are *not* reordered with older loads.
4. Loads may be reordered with older stores to different locations but *not* with older stores to the same location.
5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).
6. In a multiprocessor system, stores to the same location have a total order.
7. In a multiprocessor system, locked instructions have a total order.
8. Loads and stores are *not* reordered with locked instructions.

x86 Memory Model

- DRF programs will have sequentially consistent semantics

Java Memory Model

- Part of the *language* definition
- First version got it wrong
- Second version worth a PhD thesis
- Tension between allowing efficient optimizations by JIT and processors, and programmer understanding

Java Memory Model

- Three nice properties
  - Consistency guarantees for programmers with DRF programs
  - Guarantee for compiler writers and VM architects
  - Guarantee that programmers can use nonblocking synchronization

Java Memory Model

- DRF guarantee: Correctly synchronized programs will have sequentially consistent semantics
- Correctly synchronized IFF all sequentially consistent executions are DRF

Java Memory Model

- Different *happens-before* relation
  - Program order
  - Synchronization order
  - Volatile order
  - Thread fork/join order
  - Interrupt order
  - Finalizer order
  - Transitivity

Java Memory Model

- If non-DRF program, then what can it do?
  - Accounts for substantial part of JMM definition

What DRF Gets You

- Can ignore memory models if DRF
- Now need to *ensure* that your program is DRF
  - Static and dynamic methods
- Necessary, but often not sufficient

Other Correctness Properties

- Atomicity
  - Implementable by transactional memory
- Deterministic Parallelism
  - Generalized atomicity

Conclusion

- Memory models matter when dealing with shared memory parallel programs
- If you need performance, system details are crucial
- If you program in a data-race-free manner, many of your worries go away

References

- The Art of Multiprocessor Programming (Herlihy, Shavit)
- Java Concurrency in Practice (Goetz)
- Shared Memory Consistency Models: A Tutorial (Adve, Gharachorloo)
- Intel 64 Architecture Memory Ordering White Paper (Intel)
- The Semantics of x86 Multiprocessor Machine Code (Sarkar et al.)
- The Java Memory Model (Manson, Pugh, Adve)
- Wikipedia articles