<atomic.h> weapons

Paolo Bonzini
Red Hat, Inc.
KVM Forum 2016

The real things

- Herb Sutter’s talks
  - atomic<> Weapons: The C++ Memory Model and Modern Hardware
  - Lock-Free Programming (or, Juggling Razor Blades)
- The C11 and C++11 standards
  - N2429: Concurrency memory model
  - N2480: A Less Formal Explanation of the Proposed C++ Concurrency Memory Model

Outline

- Who ordered atomics?
- Compilers and the need for a memory model
- `qemu/atomic.h`: portable atomics in QEMU
- Future work

Outline

- Who ordered atomics?
- Compilers and the need for a memory model
- `qemu/atomic.h`: portable atomics in QEMU
- Future work

Why atomics?

- Coarse locks are simple, but scale badly
- Finer-grained locks introduce problems too
  - Not easily composable ("leaf" locks are fine, nesting can result in deadlocks)
  - Taking a lock many times is slow
- Like extremely fine-grained locks, but faster

What do atomics provide?

- Ordering of reads and writes
- Atomic compare-and-swap, like this:

```
atomic_cmpxchg(
    T *p, T expected, T desired)
{
  old = *p;
  if (*p == expected) *p = desired;
  return old;
}
```

- Everything else can be built on top of these

When to use atomics?

- When threads communicate at well-defined points
  - Example: ring buffers
- When consistency requirements are minimal
  - Example: accumulating statistics
- When complexity is easily abstracted
  - Example: synchronization primitives, data structures
- For the fast path only
  - Example: RCU, seqlock, pthread_once

Outline

- Who ordered atomics?
- Compilers and the need for a memory model
- `qemu/atomic.h`: portable atomics in QEMU
- Future work

Compiler writers are your friends

```
int i;
char *a;
a[i+4] = 1;
```

```
int n, *a;
for (int i = 0; i <= n; i++)
  a[i] = 0;
```

```
int **a;
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    a[i][j] = 42;
```

Compiler writers are your friends (but they need some help too)

assumes no overflow in i+4!

```
int i;
char *a;
a[i+4] = 1;
```

```
movb $1, 4(%rsi,%rdi)
```

infinite loop if n == INT_MAX?

```
int n, *a;
for (int i = 0; i <= n; i++)
  a[i] = 0;
```

```
int n, *a;
for (int *end = &a[n]; a <= end; )
  *a++ = 0;
```

```
int **a;
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    a[i][j] = 42;
```

```
int **a;
for (int i = 0; i < M; i++)
  for (int *row = a[i], j = 0; j < N; j++)
    row[j] = 42;
```

what if a[i][j] overwrites a[i]?

The hard truth about undefined behavior

- You don’t want the compiler to execute the program you wrote
- Most undefined behavior is obvious
- Some undefined behavior makes sense, but is hard to reason about
- Some undefined behavior seems to make no sense, but really *should* be left undefined

Sequential consistency (Lamport, 1979)

- The result of any execution is the same as if reads and writes occurred in some total order
- Operations from each individual processor are ordered the same as they appear in the program

```
static int a;
int x = ++a;
f();
return x;
```

```
static int a;

f();
return ++a;
```

Sequential consistency (Lamport, 1979)

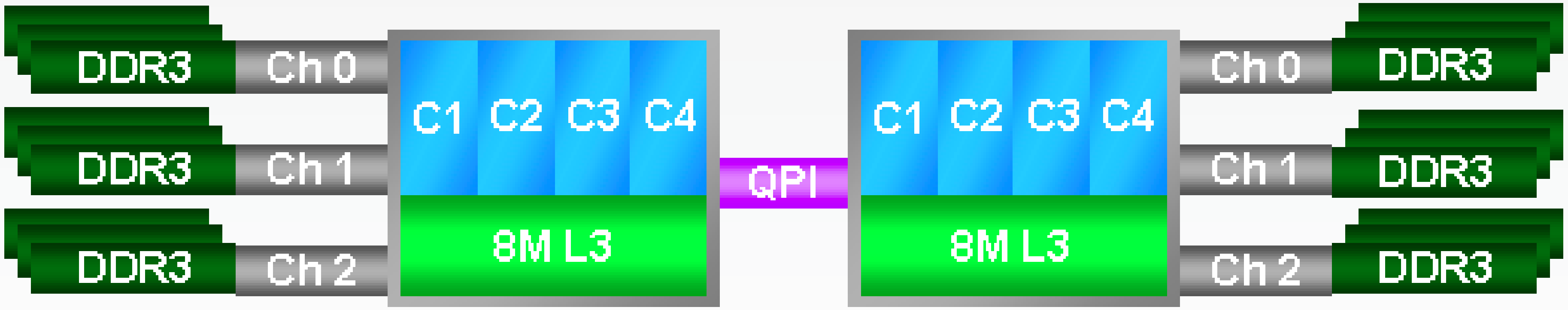
- The result of any execution is the same as if reads and writes occurred in some total order
- Operations from each individual processor are ordered the same as they appear in the program

```
long long x = 0;

// thread 1                          // thread 2
x = -1;                              printf("%lld", x);
```

Sequential consistency (Lamport, 1979)

- The result of any execution is the same as if reads and writes occurred in some total order
- Operations from each individual processor are ordered the same as they appear in the program



The C/C++ approach

- You also don’t want the processor to execute the program that you wrote
  - Processor “optimizations” can be described by rearranging loads and stores in the source code
  - Can the same tools let you reason on both compiler- and processor-level transformations?
- Union, pointers, casts: with great power comes great responsibility

The C/C++ approach

- Programs must be race-free
  - The standard precisely defines data races
  - The semantics of data races are left undefined
- If the program is “compiler-correct”, it’s also “processor-correct”
- If the program is correct, its executions are all sequentially consistent
  - … unless you turn on the guru switch

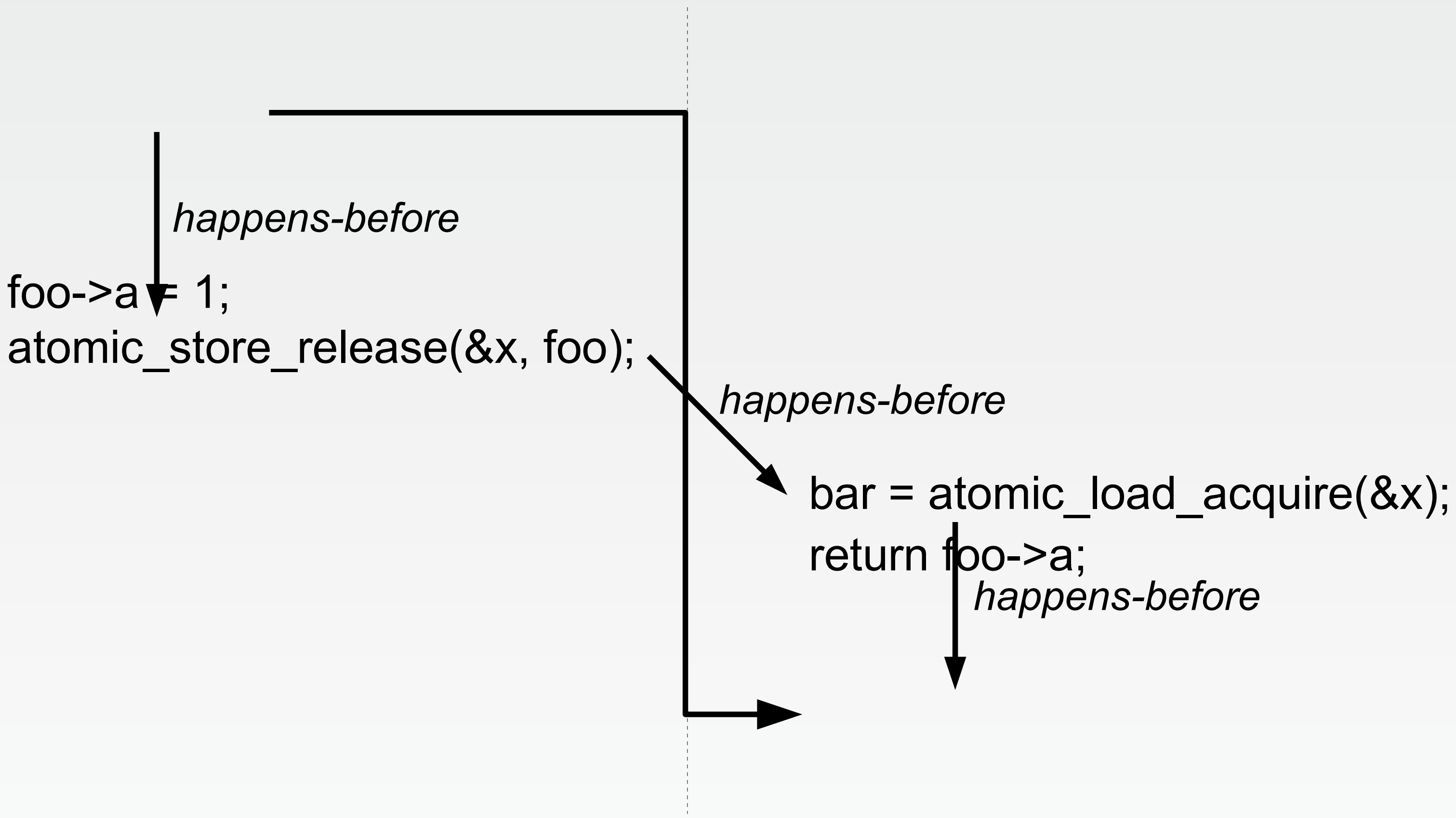
Happens-before (Lamport, 1978)

- Captures causal dependencies between events
- For any two events e1 and e2, only one is true:
  - e1 → e2 (e1 happens before e2)
  - e2 → e1 (e2 happens before e1)
  - e1 || e2 (e1 is concurrent with e2)
- **Data race:** Concurrent accesses to the same memory location, at least one a write, at least one non-atomic

More precisely...

- If a thread’s “load-acquire” sees a “store-release” from another thread, the store *synchronizes with* the load
  - The store then *happens before* the load
- Within a single thread, program order provides the happens-before relation
- Happens-before is transitive
  - Everything before the store-release *happens before* everything after the load-acquire

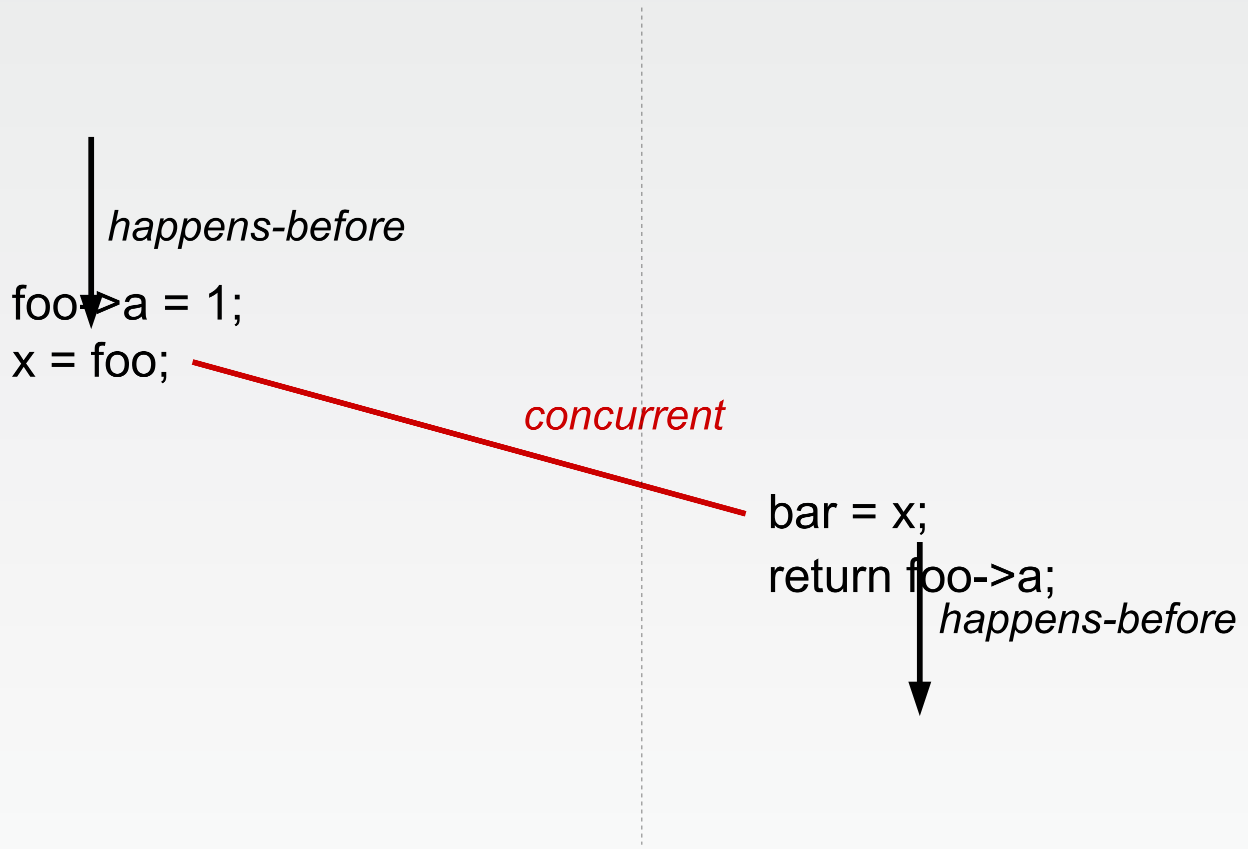
Example: data-race free, correct


```
foo->a = 1;
atomic_store_release(&x, foo);
```

```
bar = atomic_load_acquire(&x);
return foo->a;
```

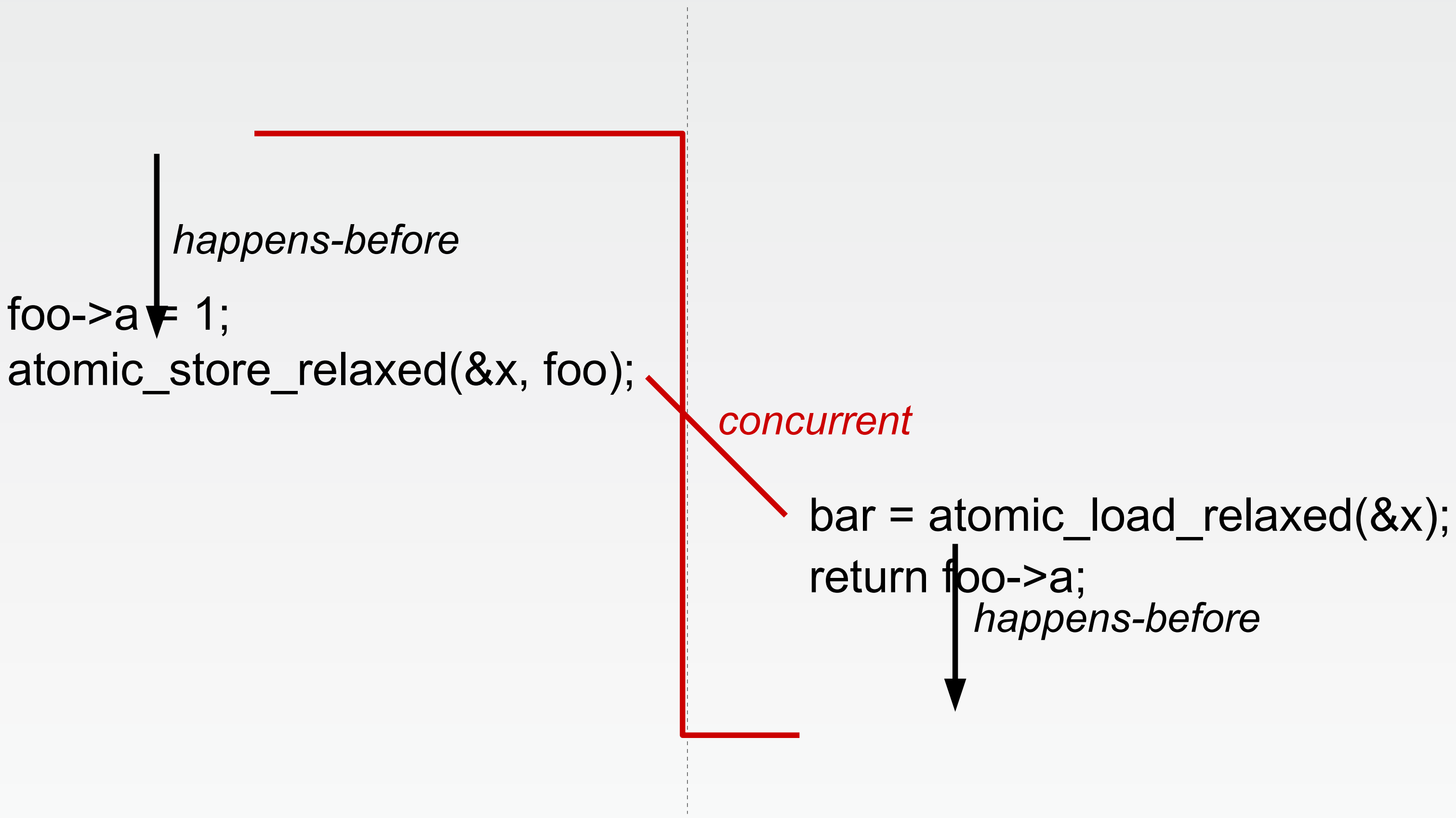
- No concurrent accesses
- No data race!

Example: data-race, undefined behavior (I)



- Concurrent non-atomic accesses, one a write
- Data race → undefined behavior!

Example: data-race, undefined behavior (II)

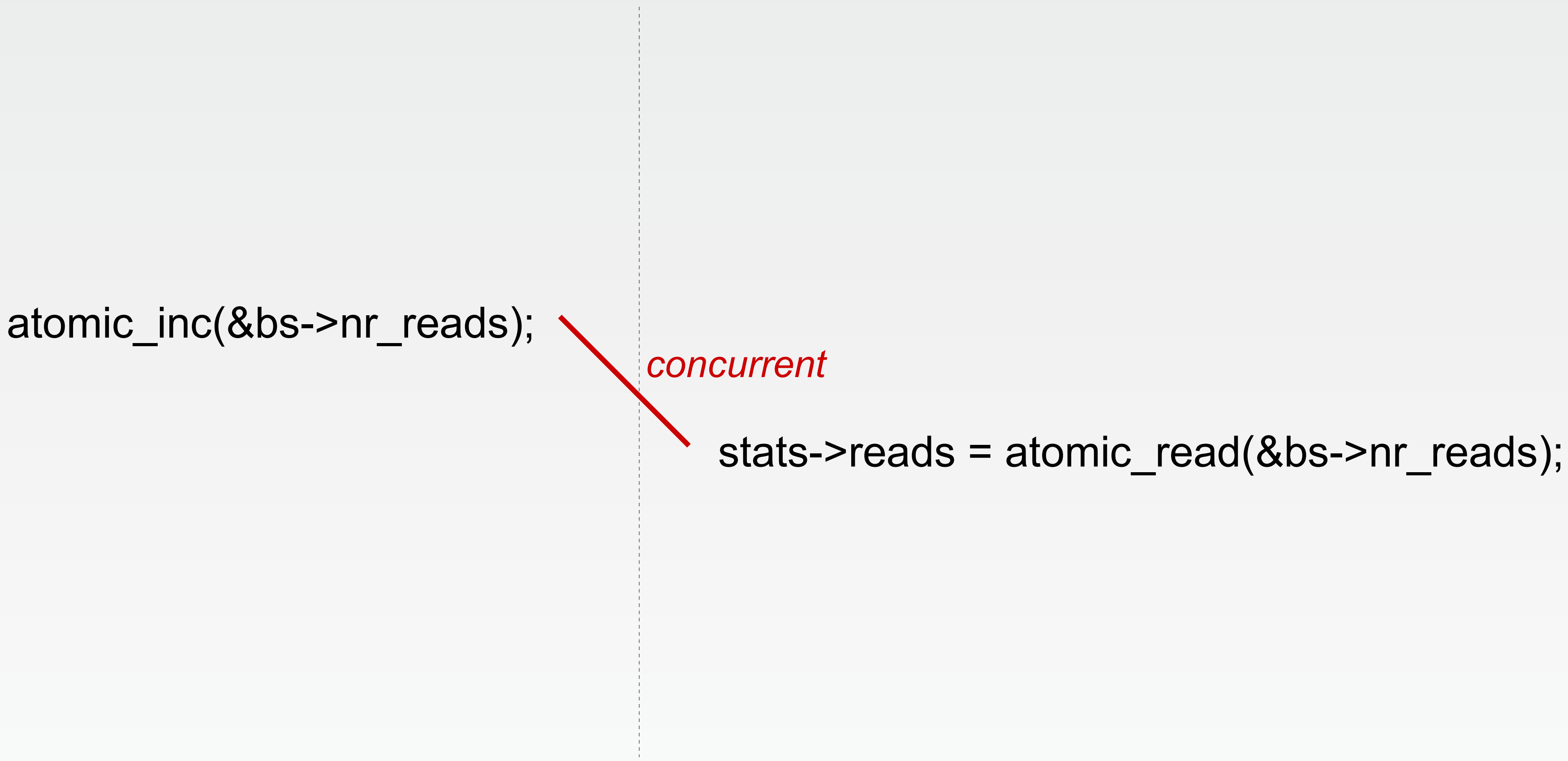


```
foo->a = 1;
atomic_store_relaxed(&x, foo);
```

```
bar = atomic_load_relaxed(&x);
return foo->a;
```

- Concurrent atomic accesses, one write
- Concurrent non-atomic accesses, one write
- No data race!
- Data race → undefined behavior!

Example: relaxed, data-race free

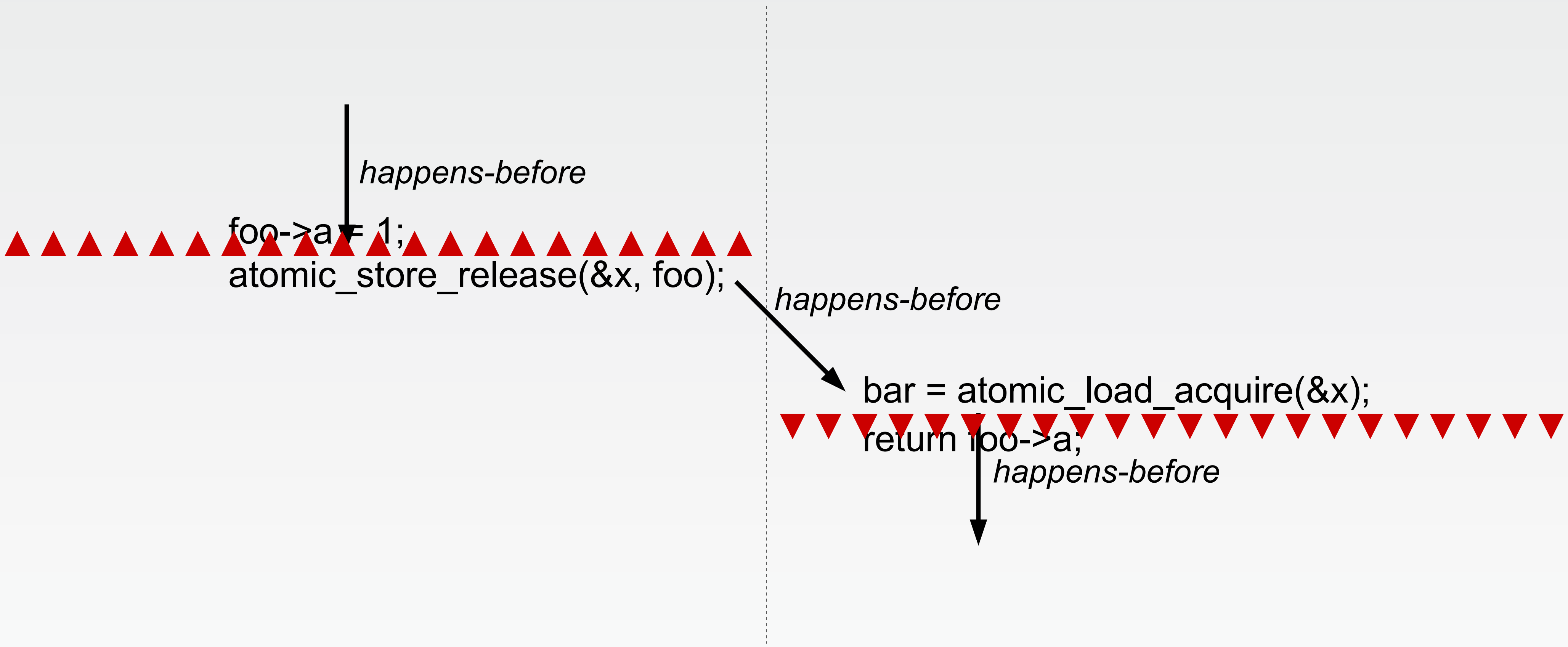


```
atomic_inc(&bs->nr_reads);
```

```
stats->reads = atomic_read(&bs->nr_reads);
```

- Concurrent atomic accesses, one a write
- No data race! But not sequentially consistent

Acquire/release as optimization barriers



```
foo->a = 1;
atomic_store_release(&x, foo);
```

```
bar = atomic_load_acquire(&x);
return foo->a;
```

Acquire and release operations

- Acquire:
  - pthread_mutex_lock
  - pthread_join
  - pthread_once
  - pthread_cond_wait

- Release:
  - pthread_mutex_unlock
  - pthread_create
  - pthread_once (first time)
  - pthread_cond_signal
  - pthread_cond_broadcast
  - pthread_cond_wait

Why atomics work

- Atomics let threads access mutable shared data without causing data races
- Atomics define happens-before across threads

- Programs that correctly use locks to prevent all data races behave as sequentially consistent
- Same for programs that do not use so-called “relaxed” atomics

Outline

- Who ordered atomics?
- Compilers and the need for a memory model
- `qemu/atomic.h`: portable atomics in QEMU
- Future work

Problems with C11 atomics

- Only supported by very recent compilers
  - Limit to what older compilers can “emulate”
- Very large API, few people can understand it
  - Start small, later add what turns out to be useful
- Some rules conflict with older usage

```
foo->bar = 1;
smp_wmb();
x = foo;
```

```
foo->bar = 1;
atomic_store(&x, foo, memory_order_release);
atomic_thread_fence(memory_order_release);
atomic_store(&x, foo, memory_order_relaxed);
```

Choosing the API

- Yes:
  - Everything seq_cst (load, store, RMW)
  - Relaxed load/store
  - RCU load/store
- Legacy:
  - Compiler barrier
  - Linux-style memory barriers
- No:
  - RMW operations other than seq_cst
- Maybe:
  - C11-style memory barriers
  - Load-acquire
  - Store-release

`qemu/atomic.h` API

- `atomic_mb_read`
  `atomic_mb_set`
- `atomic_rcu_read`
  `atomic_rcu_set`
- `atomic_read`
  `atomic_set`
- `smp_mb`
  `smp_rmb` (load-load)
  `smp_wmb` (store-store)
- `atomic_fetch_add`
  `atomic_fetch_sub`
  `atomic_fetch_inc`
  `...`
- `atomic_add`
  `atomic_sub`
  `atomic_inc`
  `...`
- `atomic_xchg`
- `atomic_cmpxchg`

Problems with portable atomics

- Less safe than C11 `stdatomic.h`

```
_Atomic int x;          int x;
x = 2;                  atomic_mb_set(&x, 2);
x += y;                 atomic_add(&x, y);
```

- Sometimes difficult to bridge C11 and “compatibility” semantics

Compatibility with older compilers

- To block optimization:
  - `volatile`
  - `asm("")` (aka `barrier();`)

  } No synchronization for multiple threads!
- `__sync_*` builtins
- If all else fails (or is too slow), `asm`

Problems with pre-C11 atomics

“[C11 atomic] accesses are guaranteed to be atomic, while volatile accesses aren't.

In the volatile case we just cross our fingers hoping that the compiler will generate atomic accesses.” (docs/atomics.txt)

Problems with pre-C11 atomics

- Only heavyweight memory barriers (`__sync_synchronize`)
- No seq-cst loads and stores
- Use asm for these

First rule of `qemu/atomic.h`

- Keep all pre-C11 hacks in there
- If really, really necessary use C11 atomics outside qemu/atomic.h
- **NEVER use asm for atomics outside qemu/atomic.h**
- **Corollary:** relaxed-atomic optimizations should only target C11 atomics

`qemu/atomic.h` API "safe" subsets

- `atomic_mb_read`
  `atomic_mb_set`
- `atomic_rcu_read`
  `atomic_rcu_set`
- `atomic_read`
  `atomic_set`
- `smp_mb`
  `smp_rmb`
  `smp_wmb`

- `atomic_fetch_add`
  `atomic_fetch_sub`
  `atomic_fetch_inc`
  `...`
- `atomic_add`
  `atomic_sub`
  `atomic_inc`
  `...`
- `atomic_xchg`
- `atomic_cmpxchg`

`qemu/atomic.h` API “less safe” subset

- `atomic_mb_read`
  `atomic_mb_set`
- `atomic_rcu_read`
  `atomic_rcu_set`
- `atomic_read`
  `atomic_set`
- `smp_mb`
  `smp_rmb`
  `smp_wmb`

- `atomic_fetch_add`
  `atomic_fetch_sub`
  `atomic_fetch_inc`
  `...`
- `atomic_add`
  `atomic_sub`
  `atomic_inc`
  `...`
- `atomic_xchg`
- `atomic_cmpxchg`

Outline

- Who ordered atomics?
- Compilers and the need for a memory model
- `qemu/atomic.h`: portable atomics in QEMU
- Future work

Choosing the API

- Yes:
  - Everything seq_cst (load, store, RMW)
  - Relaxed load/store
  - RCU load/store
- Legacy:
  - Linux-style memory barriers
- No:
  - RMW operations other than seq_cst
- Maybe:
  - C11-style memory barriers
  - Load-acquire
  - Store-release

Future work

- Modernize old code using memory barriers
  - Use atomic_read/atomic_set
  - Possibly introduce atomic_load_acquire and atomic_store_release

```
foo->bar = 1;
smp_wmb();
x = foo;
```

```
foo->bar = 1;
smp_wmb();
atomic_set(&x, foo);
```

```
foo->bar = 1;
atomic_store_release(&x, foo);
```

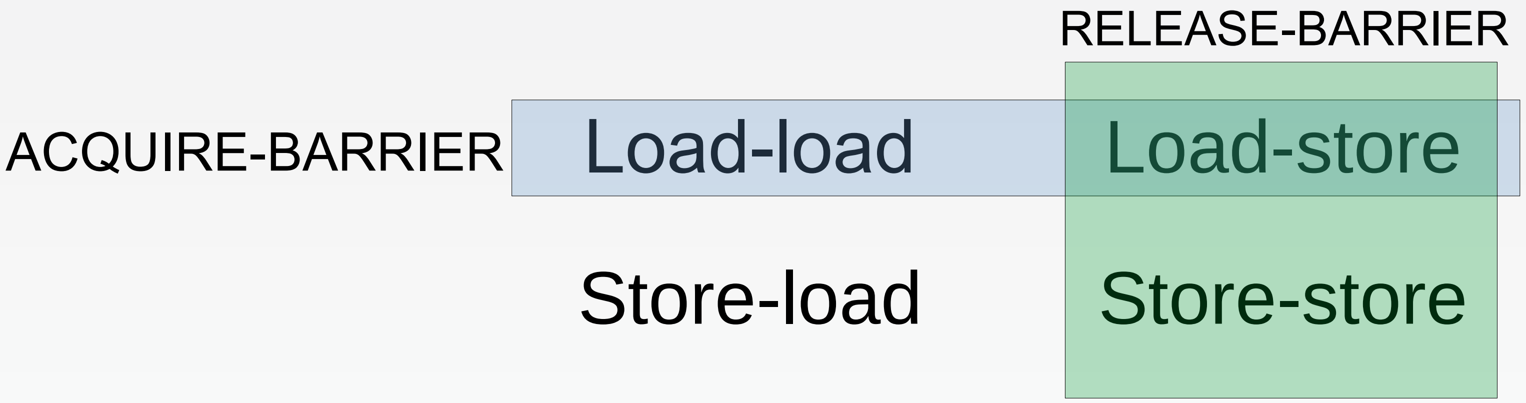
See commit 3bbf572 ("atomics: add explicit compiler fence in __atomic memory barriers")

Future work

- Modernize old code using memory barriers
  - Use atomic_read/atomic_set
  - Possibly introduce atomic_load_acquire and atomic_store_release
- Seqlock-protected fields should use atomic_read/atomic_set too

Future work

- Change Linux-style barriers to C11 barriers
  - Linux: smp_mb(), smp_rmb(), smp_wmb()
  - C11: seq-cst, acquire, release



“How to achieve [a load-store barrier] varies depending on the machine, but in practice smp_rmb()+smp_wmb() should have the desired effect.” (docs/atomics.txt)

Future work

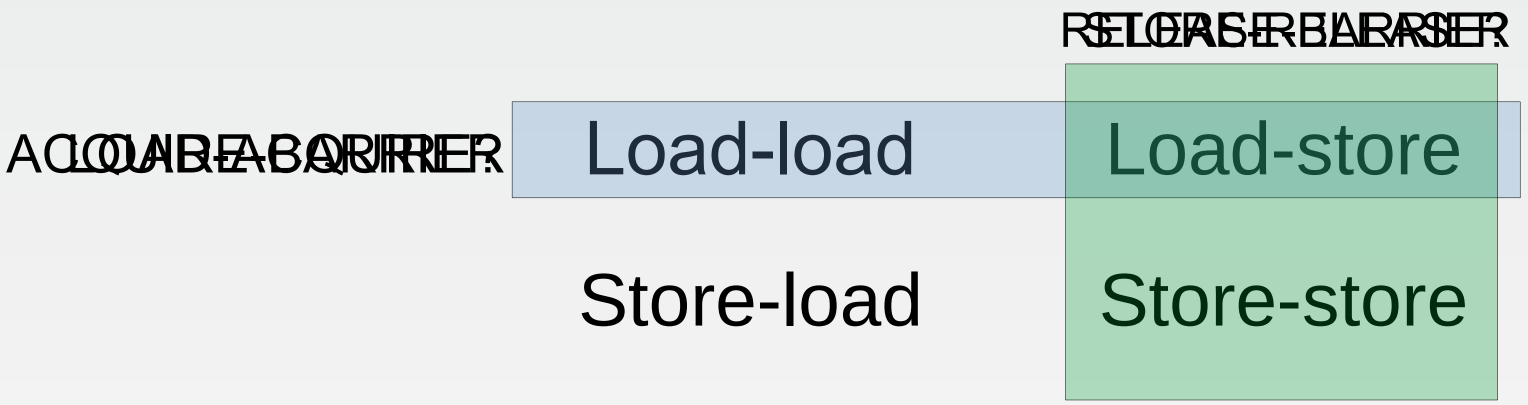
- Yes:
  - Everything seq_cst (load, store, RMW)
  - Relaxed load/store
  - C11-style memory barriers
  - Load-acquire
  - Store-release
  - Load-consume (RCU)
- No:
  - RMW operations other than seq_cst
  - Compiler barrier
  - Linux-style memory barriers

"Atomics are like a chainsaw. Everyone can learn to use one, but don't let yourself get too comfortable with it."

- Herb Sutter

Bonus material

Load-acquire/store-release vs. Acquire-barrier/release-barrier



```
store x = 1                    store z = 1
store release y = 1     ⇨      store x = 1
store z = 1                    store release y = 1        ✓
```

```
                               store z = 1
store x = 1                    store x = 1
store-store barrier     ⇨      store-store barrier        ✗
store y = 1                    store y = 1
store z = 1
```

Compiling atomics

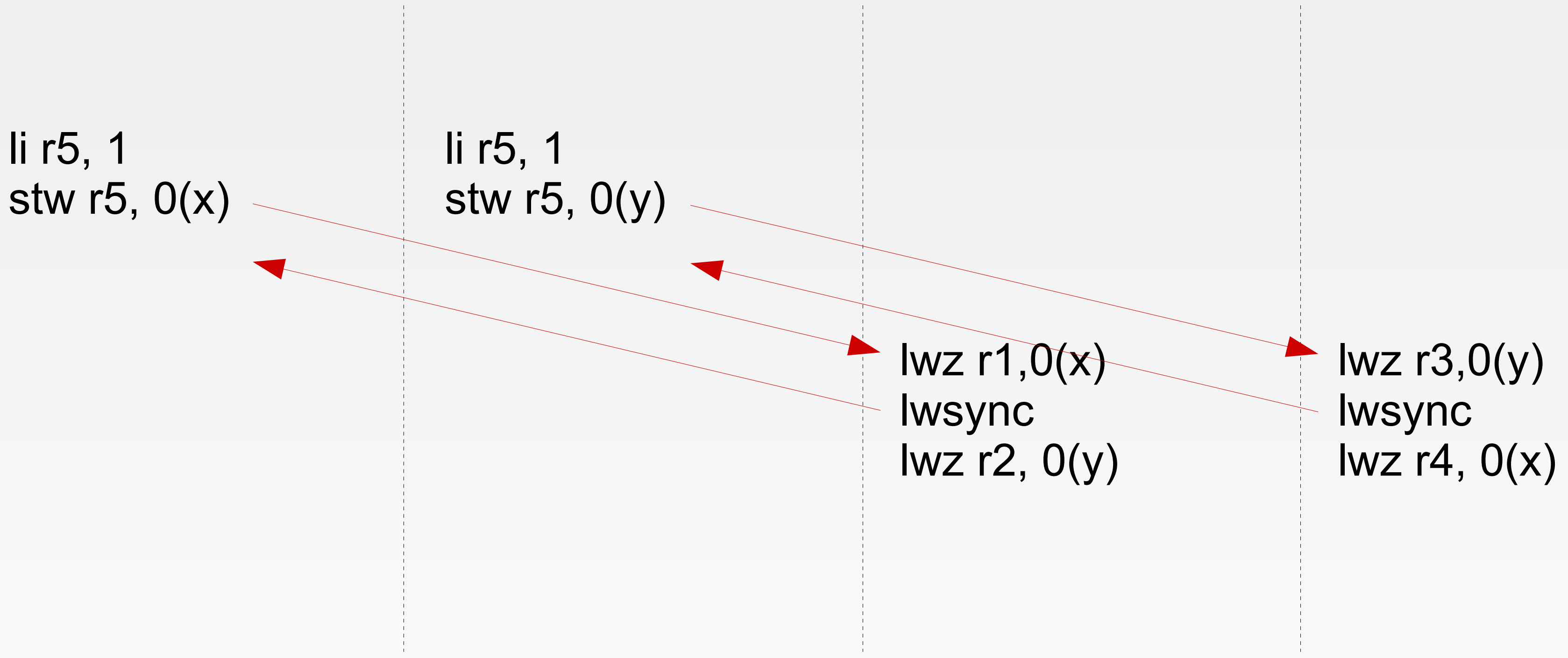
	load-consume	load-acquire	store-release	load-seqcst	store-seqcst
X86	mov	mov	mov	mov	xchg
IA64	ld.acq	ld.acq	st.rel	ld.acq	st.rel mf
ARMv7	ldr	ldr dmb	dmb ldr	ldr dmb	dmb ldr dmb
PPC	ld	ld cmp; bc; isync	lwsync st	sync ld cmp; bc; isync	sync st
AArch64	ldr	ldar	stlr	ldar	stlr

IRIW: independent reads of independent writes

x = 1;	y = 1;		
		r1 = x;	r3 = y;
		r2 = y;	r4 = x;

r1 = 1, r2 = 0, r3 = 1, r4 = 0?

IRIW and single-copy atomicity



```
li r5, 1
stw r5, 0(x)
```

```
li r5, 1
stw r5, 0(y)
```

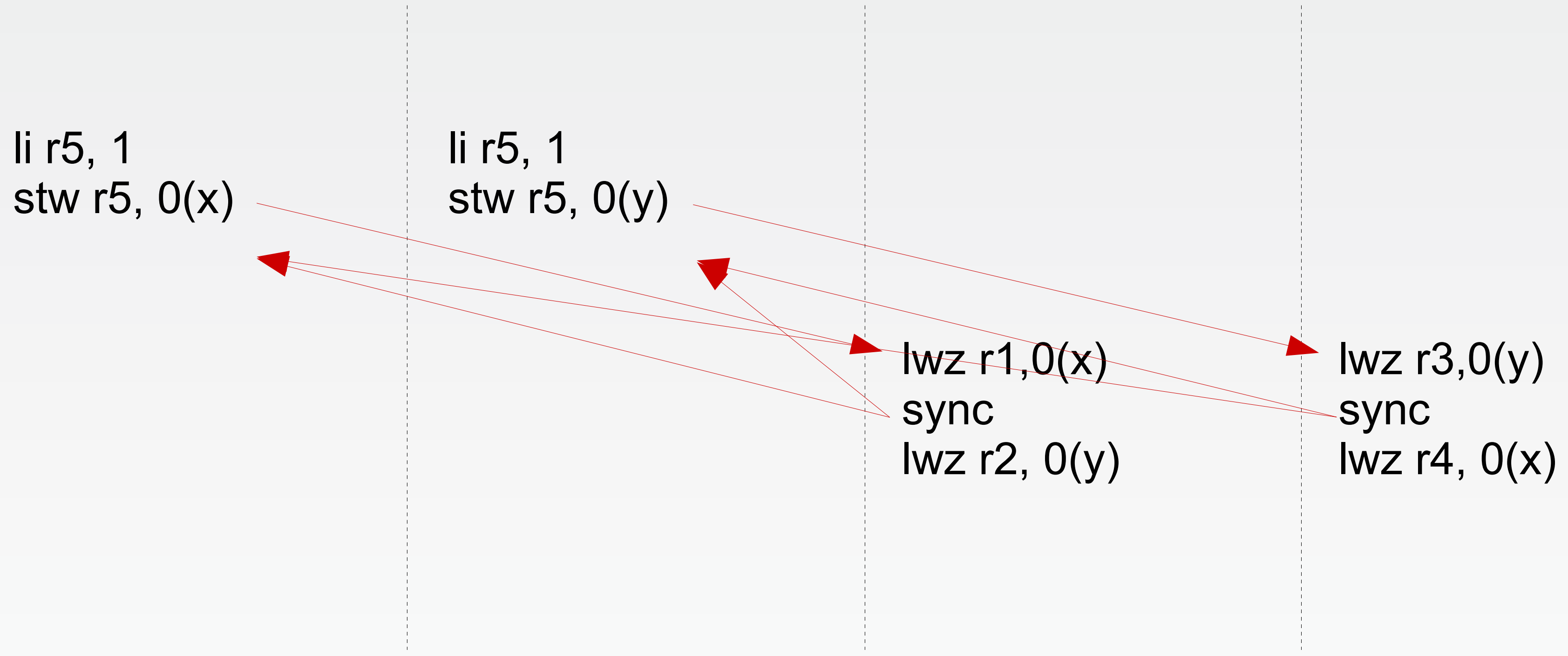
```
lwz r1,0(x)
lwsync
lwz r2, 0(y)
```

```
lwz r3,0(y)
lwsync
lwz r4, 0(x)
```

r1 = 1, r2 = 0, r3 = 1, r4 = 0?

✓ ✓ ✓ ✓

IRIW and multiple-copy atomicity



```
li r5, 1
stw r5, 0(x)
```

```
li r5, 1
stw r5, 0(y)
```

```
lwz r1,0(x)
sync
lwz r2, 0(y)
```

```
lwz r3,0(y)
sync
lwz r4, 0(x)
```

r1 = 1, r2 = 0, r3 = 1, r4 = 0?

✓ r2 = 1 ✓ r4 = 1