Memory consistency models and synchronizations

Acknowledgements

- Several of the slides in the deck are from Luis Ceze (Washington), Nima Horanmand (Stony Brook), Mark Hill, David Wood, Karu Sankaralingam (Wisconsin), Abhishek Bhattacharjee(Rutgers).

- Development of this course is partially supported by Western Digital corporations.

Locking/Mutual exclusion

Implementing a simple lock

- Shared counter/sum update example
  - Use a mutex variable for **mutual exclusion**
  - Only one processor can own the mutex
    - **Many processors may call lock(), but only one will succeed** (others block)
    - The winner executes the critical section, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

Implementing a simple lock

- Shared counter/sum update example
  - Use a mutex variable for **mutual exclusion**
  - Only one processor can own the mutex
    - **Many processors may call lock(), but only one will succeed** (others block)
    - The winner executes the critical section, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

```
1. while (lock_var != 0);!
2. lock_var = 1;!
```

Locking

- Releasing a mutex is easy
  - Just set it to 0

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - What invariant do we need?

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become ∪
  - But when it does, others will see it, too
  - What invariant do we need?

```
1. while (lock_var != 0);!
2. lock_var = 1;!
```

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically*** see that the mutex is 0 ***and*** set it to 1 (i.e., needs read-modify-write)
  - *How?*

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**
  - Many similar atomic RMW instructions such as **FetchAndAdd**, **TestAndSet**

Atomic read -modify -write instructions

- Atomic compare and exchange instruction
  - e.g., In x86-64 ISA, **cmxcg op1 (address), op2** compares value in the address op1 with value in a specific register **eax.**
  - If the values are equal then writes op2 to address pointed by op1 and sets zero flag **ZF** =1.
  - If the values are not equal then sets conditional flag **ZF=0**
  - Many similar atomic RMW instructions such as **FetchAndAdd**, **TestAndSet**

- How do you create lock/mutex with cmpxcg?

Atomic RMW for locking

```
spin_lock:
    xorl   %ecx, %ecx
    incl   %ecx          # Locked value = 1
lock_retry:
    xorl   %eax, %eax    # expected value of the lock= 0
    cmpxchgl (lock_addr), %ecx  # attempt to acquire lock
    jnz    spin_lock_retry # retry if failed (i.e., flag ZF not set)
    ret

spin_unlock:
    movl   $0,  (lock_addr)    # lock release
    ret
```

RMWs are not reordered in TSO

Earlier operation / Later operation	LD	ST	mfence	RMW
LD	NO	**YES**	NO	NO
ST	NO	NO	NO	NO
mfence	NO	NO	NO	NO
RMW	NO	NO	NO	NO

Allowed reordering of load/stores to different addresses under TSO

RMWs can be reordered in *some* weak models

Earlier operation / Later operation	LD	ST	fence	RMW
LD	**YES**	YES	NO	YES
ST	**YES**	**YES**	NO	YES
fence	NO	NO	NO	NO
RMW	YES	YES	NO	NO

Allowed reordering of load/stores to different addresses under weaker memory models

How to implement RMWs?

- Assume TSO

How to implement RMWs?

- Assume TSO
  - Flush/drain write buffer
  - Don’t allow load/stores after the RMW to execute

How to implement RMWs?

- Assume TSO
  - Flush/drain write buffer
  - Don’t allow load/stores after the RMW to execute
  - Get cache block containing the RMW variable with exclusive permission

How to implement RMWs?

- Assume TSO
  - Flush/drain write buffer
  - Don't allow load/stores after the RMW to execute
  - Get cache block containing the RMW variable with exclusive permission
  - Lock the cache line until the entire operation is complete – don't respond to coherence invalidations

How to implement RMWs?

- Assume TSO
  - Flush/drain write buffer
  - Don't allow load/stores after the RMW to execute
  - Get cache block containing the RMW variable with exclusive permission
  - Lock the cache line until the entire operation is complete – don't respond to coherence invalidations
- How would the above implementation change for a weak memory model?

RMWs are costly

- Exclusive permission → invalidate sharers
  - Coherence would ensure single writer
- A lot coherence traffic:
  - If there are multiple sharers all trying do RMWs on a sherd variable (e.g., trying to take lock)

RMWs are costly

- Exclusive permission → invalidate sharers
  - Coherence would ensure single writer
- A lot coherence traffic:
  - If there are multiple sharers all trying do RMWs on a sherd variable (e.g., trying to take lock)
- How to reduce invalidation traffic?

RMWs are costly

- Exclusive permission → invalidate sharers
  - Coherence would ensure single writer
- A lot coherence traffic:
  - If there are multiple sharers all trying do RMWs on a sherd variable (e.g., trying to take lock)
- How to reduce invalidation traffic?
  - Spin on local cached copy in each sharer
    - Do normal load first
    - If it changes value, then only do RMW

RMWs are costly

- Exclusive permission → invalidate sharers
  - Coherence would ensure single writer
- A lot coherence traffic:
  - If there are multiple sharers all trying do RMWs on a sherd variable (e.g., trying to take lock)
- How to reduce invalidation traffic?
  - Spin on local cached copy in each sharer
    - Do normal load first
    - If it changes value, then only do RMW
  - Only one of sharer will succeed in RMW
    - Who decides who wins the race?

Alternative to RMWs: Load -linked, Store -conditional

- Typically found in RISC ISAs (e.g., ARM, MIPS)
  - Two instructions, not one, but works as pair

Alternative to RMWs: Load -linked, Store -conditional

- Typically found in RISC ISAs (e.g., ARM, MIPS)
  - Two instructions, not one, but works as pair
- Load linked: `LL rt, offset(rs)`
- Store conditional: `SC rt, offset(rs)`
  - Succeeds if location not changed since the LL
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt

Alternative to RMWs: Load-linked, Store-conditional

- Typically found in RISC ISAs (e.g., ARM, MIPS)
  - Two instructions, not one, but works as pair
- Load linked: `LL rt, offset(rs)`
- Store conditional: `SC rt, offset(rs)`
  - Succeeds if location not changed since the LL
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- If the store conditional succeeds, then the atomicity is guaranteed.

Locking with LL/SC

```
m = 0;
mutex_lock(int *m) {
   test_and_set:
                LI $t0, 1
                LL $t1, 0($a0)
                BNEZ $t1, test_and_set
                SC $t0, 0($a0)
                BEQZ $t0, test_and_set //Atomicity is broken
}

mutex_unlock(int *m) {
        *m = 0;
}
```

How does H/W implement LL/SC?

- A link register is introduced (new)
  - Stores the address specified in the LL instruction
- If an Interrupt comes or if invalidation for cache block address matching the link register arrives
  - Clear the link register
  - Same idea as snooping, listening for a write on the address of the link register
- Store conditional fails if the link register is cleared

Back to consistency models

Release consistency model

- A type of weak memory model
  - Typically implemented in ARM, IBM Power
- Two new key operations for synchronization
  - Acquire (lock) → Taking the lock
  - Release (lock) → Unlock

Release consistency model

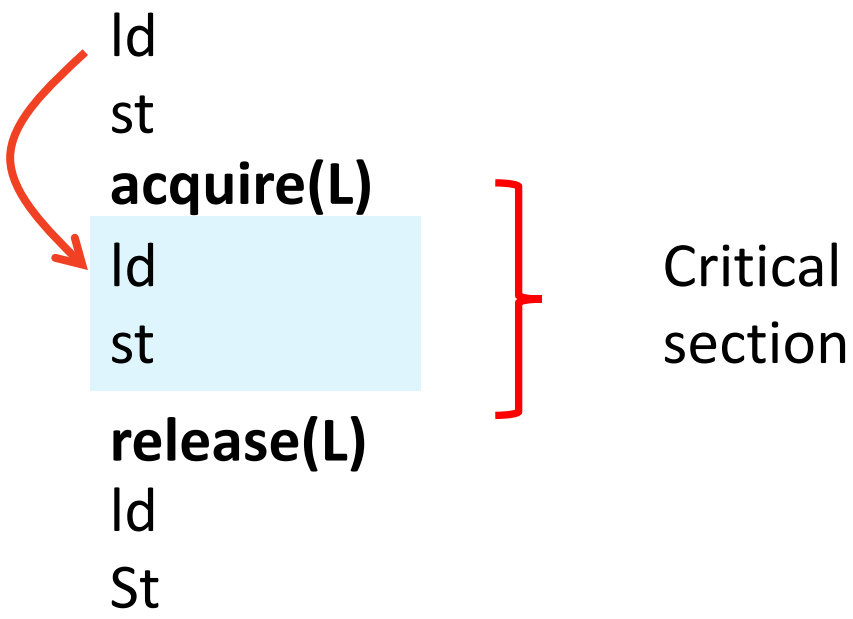
- A type of weak memory model
  - Typically implemented in ARM, IBM Power
- Two new key operations for synchronization
  - Acquire (lock) → Taking the lock
  - Release (lock) → Unlock
- A acquire/release pair defines critical section in a multi-threaded code
  - Critical section is part of code that can be executed by only one thread at any given time
  - Ensures mutual exclusion

Release consistency model

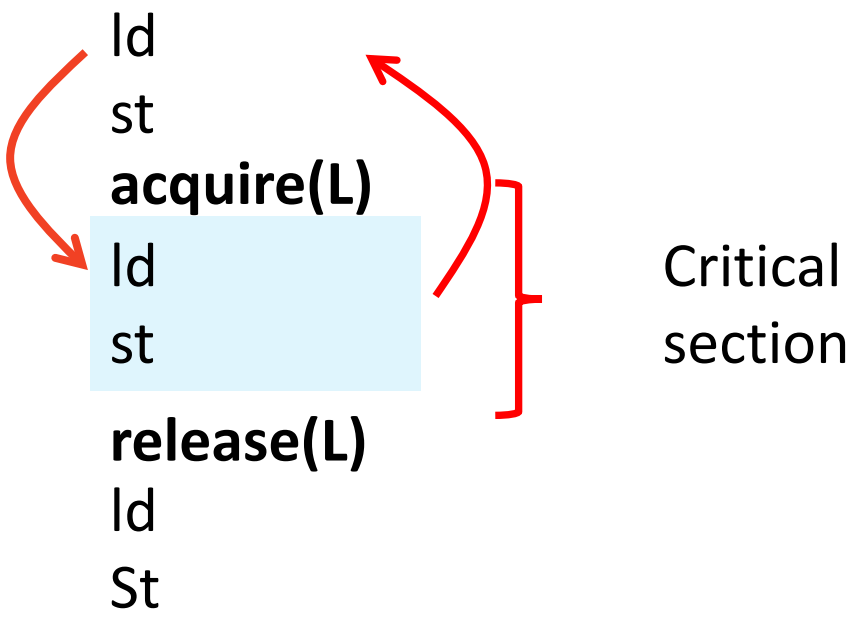
ld
st
acquire(L)
ld
st
release(L)
ld
St

Critical section

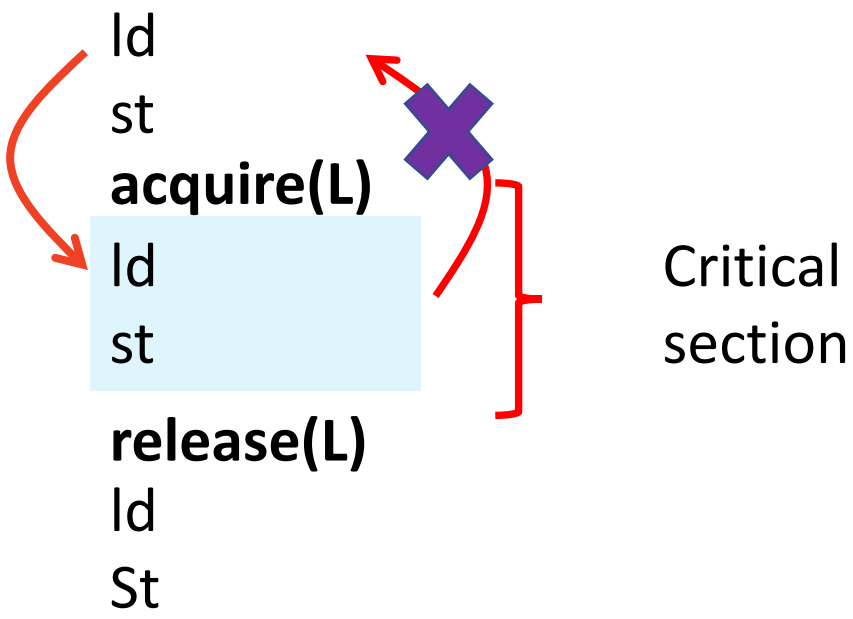
Release consistency model



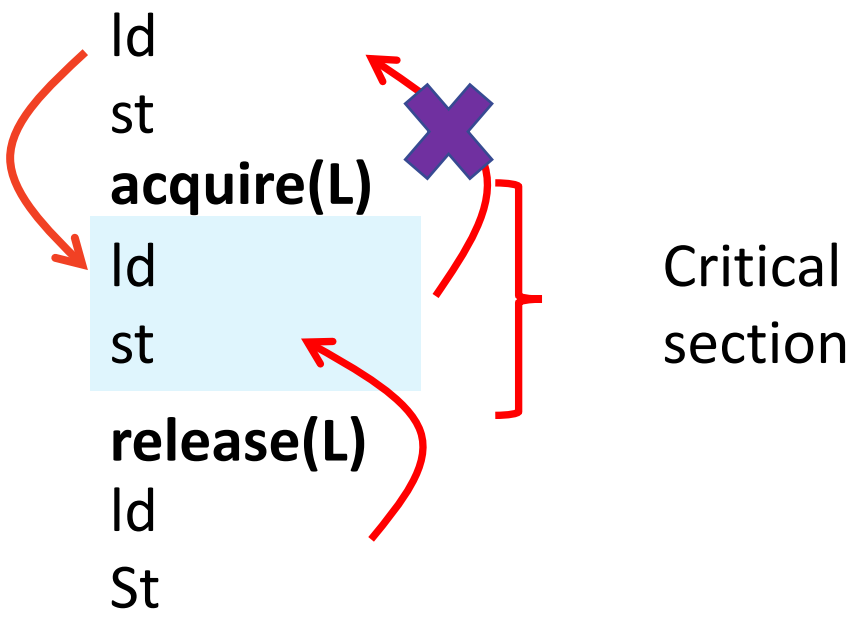
Release consistency model



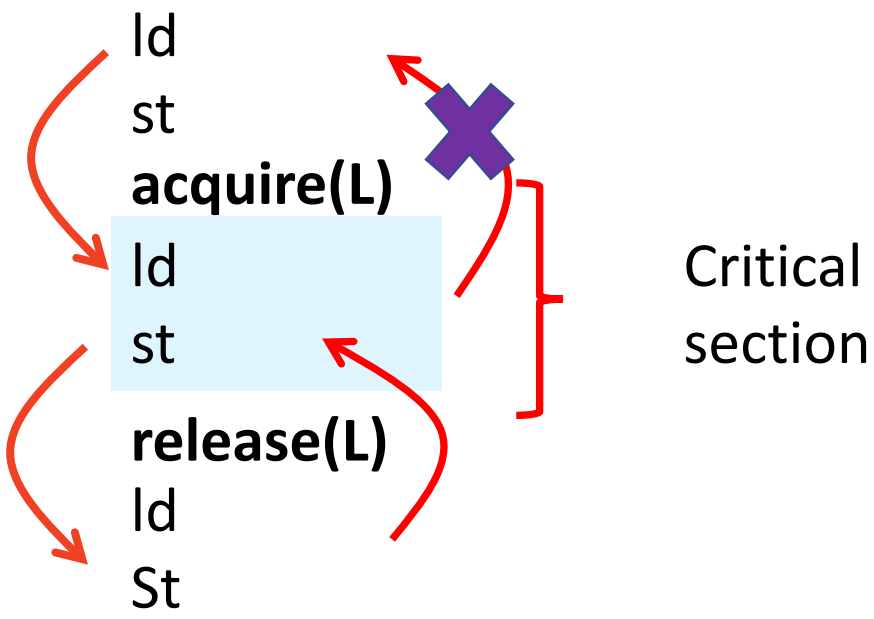
Release consistency model



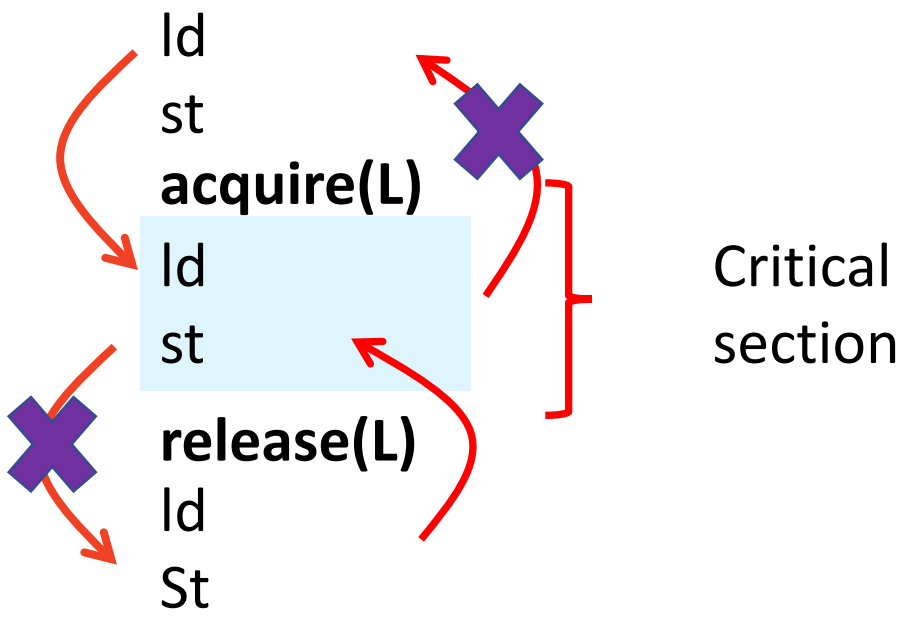
Release consistency model



Release consistency model



Release consistency model



Release consistency model

Earlier operation / Later operation	LD	ST	Acquire	Release
LD	YES	YES	NO	YES
ST	YES	YES	NO	YES
Acquire	YES	YES	NO	NO
Release	NO	NO	NO	NO

Role of compilers

- We do not write programs in assembly language
- So compilers/runtimes need to be aware of memory models
  - Memory models restricts what code movement/optimization a compiler can do
  - For example, a SC abiding compiler cannot reorder loads/stores.
  - Cannot typically do common subexpression elimination

Role of compilers

- We do not write programs in assembly language
- So compilers/runtimes need to be aware of memory models
  - Memory models restricts what code movement/optimization a compiler can do
  - For example, a SC abiding compiler cannot reorder loads/stores.
  - Cannot typically do common subexpression elimination
- What happens if a compiler guarantees only release consistency but the hardware support TSO?
  - What consistency model the programmer will see?

Memory models for modern languages

- C++ (from C++ 11) and Java has its memory model
  - It's called sequential consistency for **data-race-free** program
  - In short, it's called **SC-for-DRF**

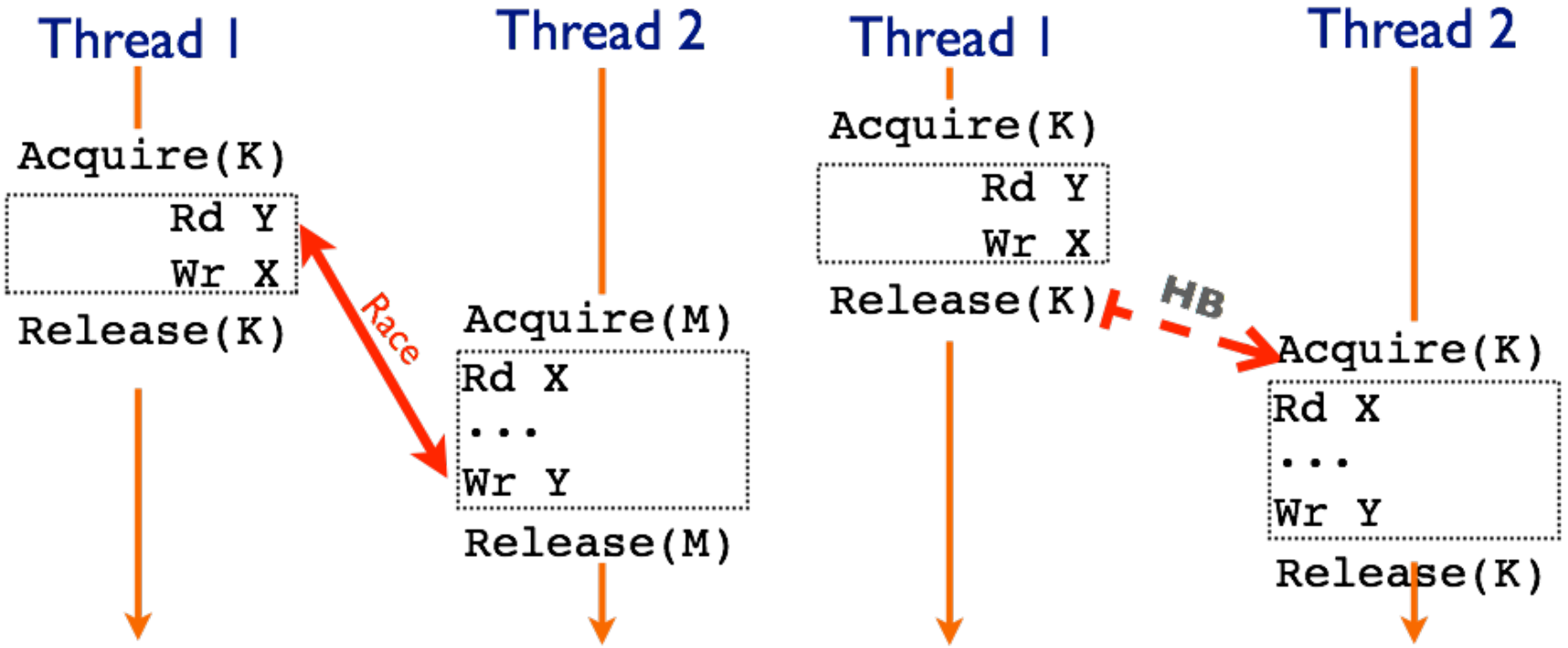
- Defines two types of memory accesses
  - Synchronization operations
    - e.g., acquire, release
  - Data operation
    - All operations that are not synchronization operation
  - Memory allocations are tagged if they are atomic

What is a data race?

- Two accesses from **different threads**; at **least one a write**; accessing the **same location**; without explicit happens-before **ordering via synchronization**.

What is a data race?

- Two accesses from **different threads**; at **least one a write**; accessing the **same location**; without explicit happens-before **ordering via synchronization**.



Data race and Race condition

- Data Races != Race Conditions

- Race Condition
  - Any timing error in the program
  - Due to events, device interaction/interrupts, thread interleaving, …

- Data races are not sufficient nor necessary for a race condition
  - Data race just a good symptom for possible race condition

SC-for -data race free (DRF)

SC-for -data race free (DRF)

- An SC execution is data-race-free (DRF) if no data operations can race.
- A program is DRF if all its SC executions are DRF.

SC-for -data race free (DRF)

- An SC execution is data-race-free (DRF) if no data operations can race.
- A program is DRF if all its SC executions are DRF.
- A memory consistency model supports “SC for DRF programs” **if all executions of all DRF programs are SC executions**.

SC-for -data race free (DRF)

- An SC execution is data-race-free (DRF) if no data operations can race.
- A program is DRF if all its SC executions are DRF.
- A memory consistency model supports "SC for DRF programs" **if all executions of all DRF programs are SC executions**.
  - For DRF program, the system (compiler, h/w) should guarantee *appearance* of Sequential consistency

SC-for -data race free (DRF)

- An SC execution is data-race-free (DRF) if no data operations can race.
- A program is DRF if all its SC executions are DRF.
- A memory consistency model supports "SC for DRF programs" **if all executions of all DRF programs are SC executions**.
  - For DRF program, the system (compiler, h/w) should guarantee *appearance* of Sequential consistency
- No guarantee (undefined) for programs with data races.
- Note: DRF does not mean that output is deterministic.

SC-for -data race free (DRF)

- Put onus on programmer to write properly synchronized program.
  - Or else **no** guarantee of any sorts.

SC-for -data race free (DRF)

- Put onus on programmer to write properly synchronized program.
  - Or else **no** guarantee of any sorts.

- Allows the compiler and the hardware (each core) re-order any load/stores to different addresses but no re-ordering with synchronization operations
  - If there is no data race, executions cannot be differentiated from an execution under SC
  - The h/w implementation could be a relaxed model

Why SC -for -DRF make sense?

Thread 0 (Core 0)

```
acquire(L)
        st x 1
        st y 1
release(L)
```

Thread 1 (Core 1)

```
acquire(L)
        ld r1 y
        ld r2 x
release(L)
```

Why SC -for -DRF make sense?

Thread 0 (Core 0)

```
acquire(L)
        st x 1
        st y 1
release(L)
```

Thread 1 (Core 1)

```
acquire(L)
        ld r1 y
        ld r2 x
release(L)
```

- What values of x,y possible under SC?
  - Assume initial value being 0

Why SC -for -DRF make sense?

Thread 0 (Core 0)

```
acquire(L)
        st x 1
        st y 1
release(L)
```

Thread 1 (Core 1)

```
acquire(L)
        ld r1 y
        ld r2 x
release(L)
```

- What values of x,y possible under SC?
  - Assume initial value being 0
- What values are possible under SC-for-DRF?

What SC -for -DRF buys?

- A *lot* of freedom to compiler and hardware
  - H/W: Coalescing write buffer, load-load reordering (major source of MLP)
  - Compiler: loop-inv code motion, common subexpression elimination, etc.
- Pretty much can do whatever reordering as long as it *does not cross synchronization point*

Example SC -for -DRF compliant program in C++ (C++11)

```
                A=0   FLAG=0
C 0                              C 1
ST A 1;                          L1: LD r1 FLAG
ST FLAG 1;                       If  r1 == 0; JMP L1;// spin lock
                                    LD r2 A;
```

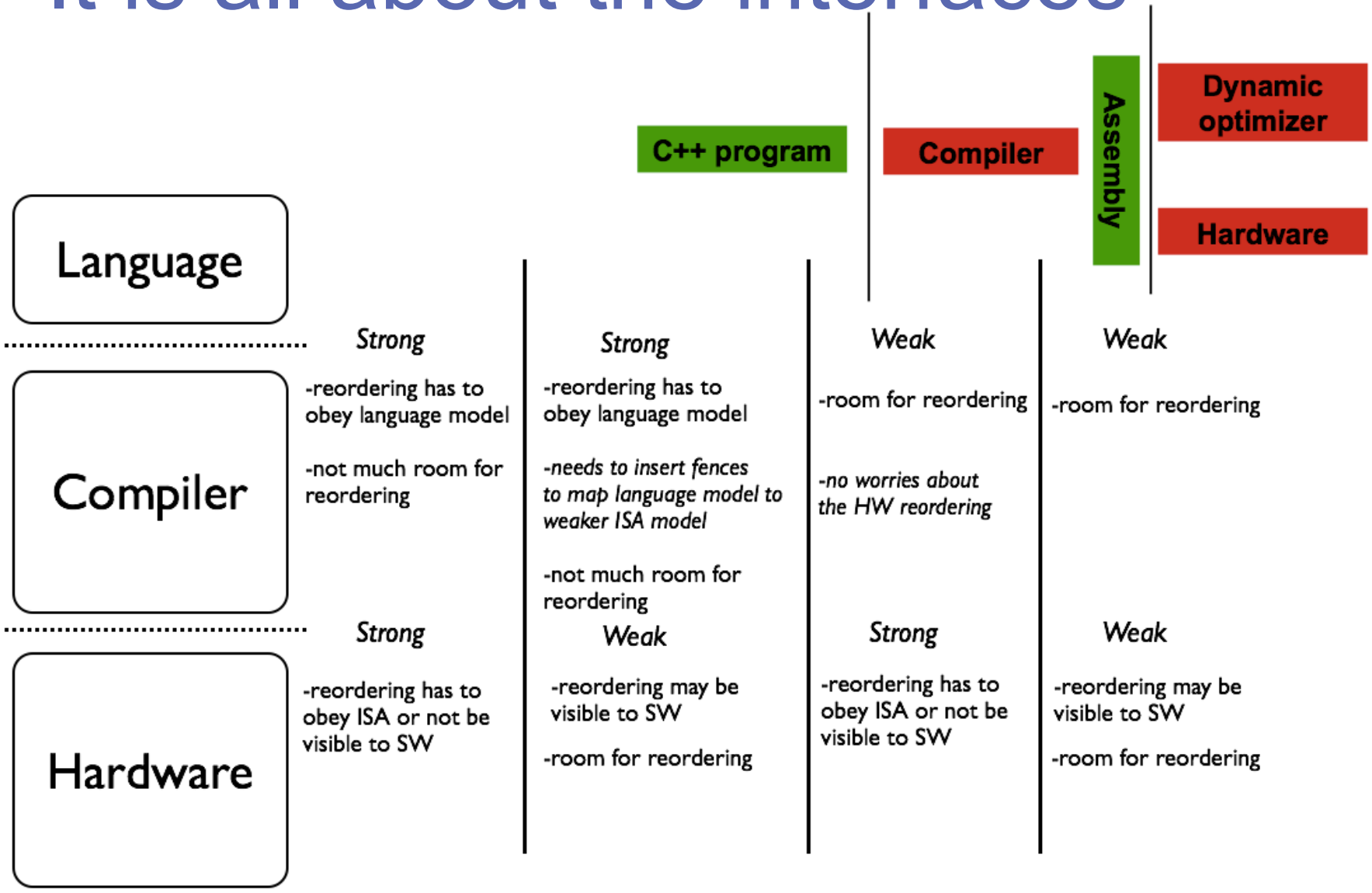
Example SC -for -DRF compliant program in C++ (C++11)

```
                A=0   FLAG=0
 C 0                          C 1
 ST A 1;                      L1: LD r1 FLAG
 ST FLAG 1;                   If  r1 == 0; JMP L1;// spin lock
                                 LD r2 A;
```

```
                int A(0); atomic_int  FLAG(0);
C 0                              C 1
A=1;                             while(!FLAG.load(acquire)){};
FLAG.store(1, release);             cout<<A<<endl;
```

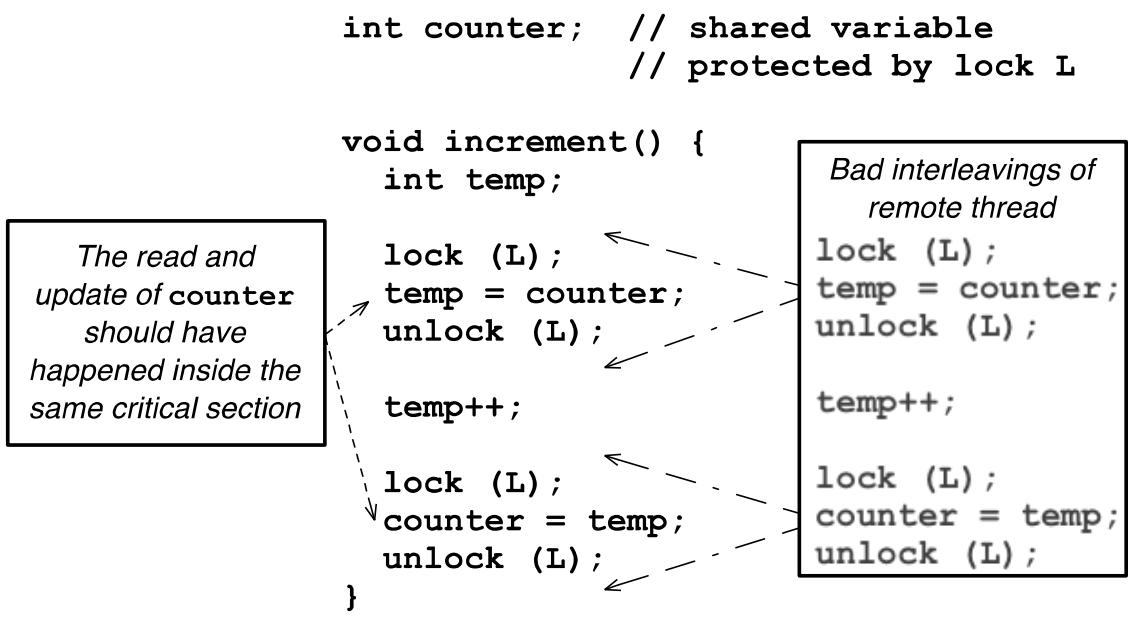
- a DRF version in C++ semantics

It is all about the interfaces



Quick aside: General concurrency errors…

- Does free of data-races mean "correct" concurrency-wise?



» *Atomicity violations* aren't races necessarily

Writing "correct" multi - threaded programs hard

- Debugging is even harder
  - Bug may occur only under a specific interleaving of load/stores → non-deterministic execution
  - Errors may not be reproducible ! How to debug?

Writing "correct" multi - threaded programs hard

- Debugging is even harder
  - Bug may occur only under a specific interleaving of load/stores → non-deterministic execution
  - Errors may not be reproducible ! How to debug?

Writing "correct" multi - threaded programs hard

- Debugging is even harder
  - Bug may occur only under a specific interleaving of load/stores → non-deterministic execution
  - Errors may not be reproducible ! How to debug?

- Active research since advent of multi-cores
  - Data race detectors
    - In hardware
    - In software → either static (looking at code) or dynamic (during execution)
  - Atomicity violation detectors
  - Deterministic re-execution
    - Reproduce a previous execution –helpful for debugging
    - Log some important interleaving information