Concepts and Models of Parallel and Data-centric Programming

Shared Memory VII

Lecture, Summer 2020

Dr. Christian Terboven <terboven@itc.rwth-aachen.de>

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Outline

Memory Consistency & Atomicity

Why do we need memory consistency?

Core C1	Core C2	Notes
S1: STORE data = NEW		/* Init: data = 0, flag != SET */
S2: STORE flag = SET	L1: LOAD r1 = flag	/* L1 & B1 can repeat many times */
	B1: if (r1 != SET) goto L1	
	L2: LOAD r2 = data	

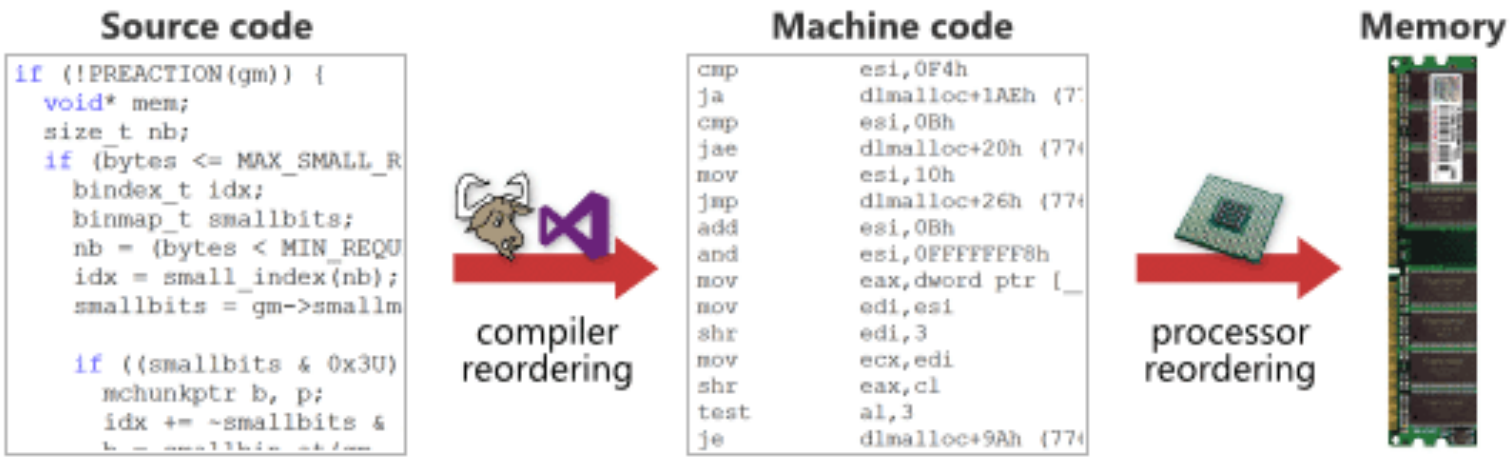
Why do we need memory consistency?

Core C1	Core C2	Notes
S1: STORE data = NEW		/* Init: data = 0, flag != SET */
S2: STORE flag = SET	L1: LOAD r1 = flag	/* L1 & B1 can repeat many
	B1: if (r1 != SET) goto L1	times */
	L2: LOAD r2 = data	

- Is **r2** always set to **NEW**?
  - Intuitively: Yes, because **NEW** is stored in **data** before **SET** is stored in **flag**.
  - Reality (for modern hardware and compilers): No guarantee without memory consistency model.

Memory Operation Reordering (1)

- Memory operations can be reordered
  - During compilation step: Optimization of data accesses
  - During execution: Processor (core) optimizes usage of pipelines, caches etc.



Source: https://preshing.com/20120930/weak-vs-strong-memory-models/

Memory Operation Reordering (2)

Core C1	Core C2	Notes
S1: STORE data = NEW		/* Init: data = 0, flag != SET */
S2: STORE flag = SET	L1: LOAD r1 = flag	/* L1 & B1 can repeat many times */
No dependencies between S1 and S2, reordering possible.	B1: if (r1 != SET) goto L1	
	L2: LOAD r2 = data	

- Memory operations cannot be arbitrarily reordered (data dependencies)
- **But:** As long as behavior of (isolated) single-threaded execution is not changed: Reordering by compiler and hardware allowed
  - Memory operations are reordered only with a local view on each core, accesses from other cores not considered
- Thus: **r2 = 0** could be valid outcome of execution

Memory Operation Reordering (2)

Core C1	Core C2	Notes
S2: STORE flag = SET S1: STORE data = NEW No dependencies between S1 and S2, reordering possible.	L1: LOAD r1 = flag B1: if (r1 != SET) goto L1 L2: LOAD r2 = data	/* Init: data = 0, flag != SET */ /* L1 & B1 can repeat many times */

- Memory operations cannot be arbitrarily reordered (data dependencies)
- **But:** As long as behavior of (isolated) single-threaded execution is not changed: Reordering by compiler and hardware allowed
  - Memory operations are reordered only with a local view on each core, accesses from other cores not considered
- Thus: **r2 = 0** could be valid outcome of execution

Another Example

Core C1	Core C2	Notes
S1: STORE x = NEW L1: LOAD r1 = y	S2: STORE y = NEW L2: LOAD r2 = x	/* Init: x = 0, y = 0 */

Another Example

Core C1	Core C2	Notes
S1: STORE x = NEW L1: LOAD r1 = y	S2: STORE y = NEW L2: LOAD r2 = x	/* Init: x = 0, y = 0 */

- What are possible outcomes for the tuple (r1, r2)?
  - (0, NEW) for execution S1, L1, S2, L2
  - (NEW, 0) for execution S2, L2, S1, L1
  - (NEW, NEW) for execution S1, S2, L1, L2

Another Example

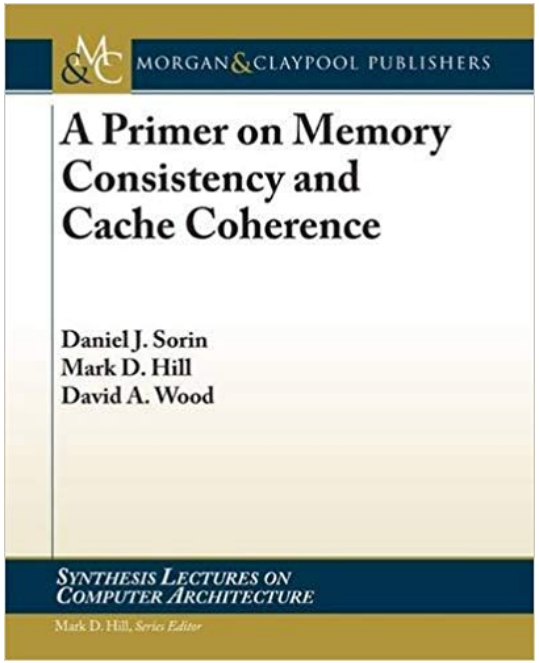
Core C1	Core C2	Notes
S1: STORE x = NEW	S2: STORE y = NEW	/* Init: x = 0, y = 0 */
L1: LOAD r1 = y	L2: LOAD r2 = x	

- What are possible outcomes for the tuple (r1, r2)?
  - (0, NEW) for execution S1, L1, S2, L2
  - (NEW, 0) for execution S2, L2, S1, L1
  - (NEW, NEW) for execution S1, S2, L1, L2
- Is (0, 0) a possible outcome?
  - Yes, reordering of S1 / L1 and S2 / L2 possible (no dependencies)
  - (0, 0) for execution L1, S2, L2, S1
  - x86 architectures allow this kind of memory reordering

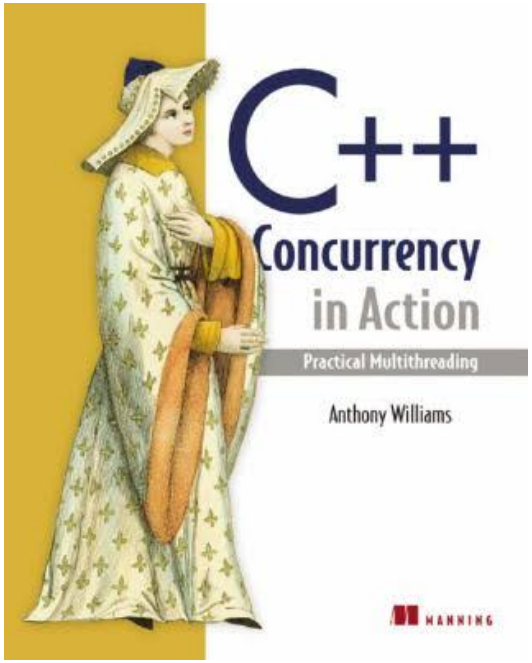
Where are memory models required for developers?

- Two possible ways of achieving consistent shared memory accesses between threads
- **Lock-based codes**
  - Use locks to avoid concurrent accesses to same memory addresses
    → Program can deadlock or livelock
- **Lock-free codes**
  - Use atomics to coordinate concurrent accesses to same memory addresses
    → Program can never lock up
  - Problem: Memory operation reordering can influence correctness of algorithms
  - Clear reasoning on memory consistency required

Literature



Sorin, Daniel J., Mark D. Hill, and David A. Wood. "A Primer on Memory Consistency and Cache Coherence." *Synthesis Lectures on Computer Architecture* 6.3 (2011): 1-212.



Williams, Anthony. C++ concurrency in action. Manning, 2017.

- **Disclaimer:** Memory models are a very complicated topic, not covered in all aspects in this lecture.

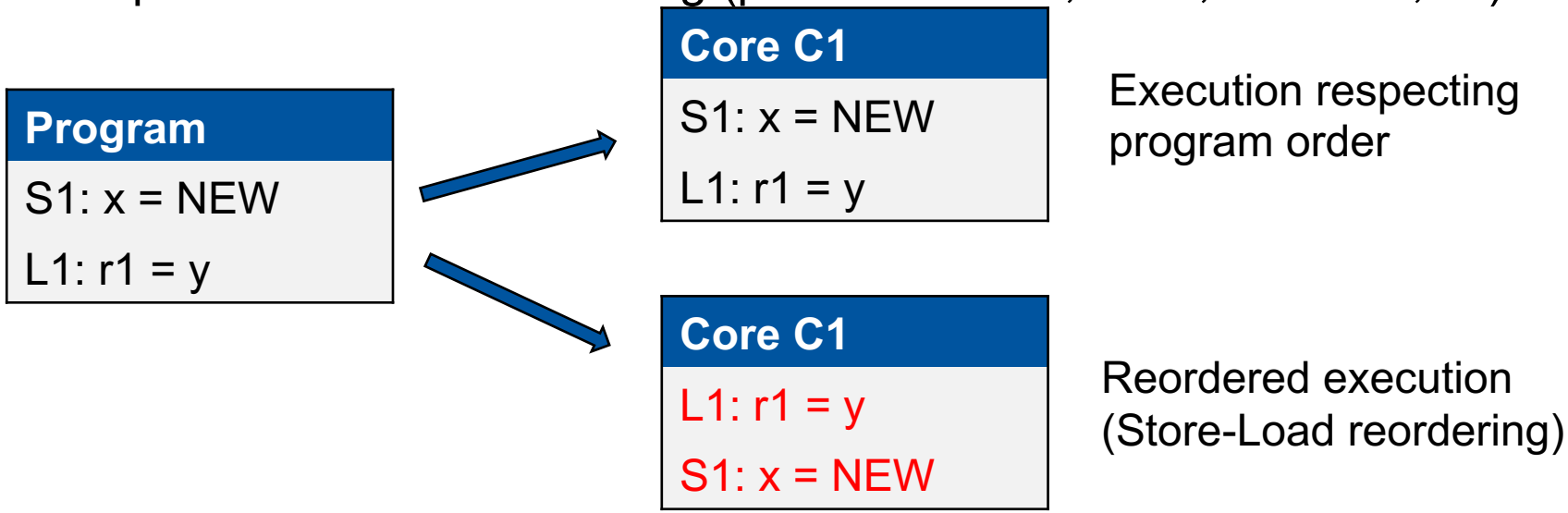
Memory Consistency Models

- Memory consistency models (short: memory models)
  - Specification of allowed behavior of multithreaded programs executing with shared memory
  - Consists of rules defining allowed (re-)ordering of Loads and Stores in memory
  - Avoids unintended effects of memory operations reordering

- Memory models separate multithreaded executions in **valid** (following rules) and **invalid** (violating rules) executions

Kinds of Reorderings

- Four different kinds of reorderings possible:
  - **LoadLoad** reordering: Load instruction reordered before / after another
  - **LoadStore** reordering: Store instruction reordered before load instruction
  - **StoreLoad** reordering: Load instruction reordered before store instruction
  - **StoreStore** reordering: Store instruction reordered before / after another
- Example: Store-Load reordering (possible in x86, ARM, POWER, …)



Sequential Consistency (SC)

- Sequential consistency (SC): Enforce all orderings of memory operations
  - Forbids all four types of reorderings for any memory operation

- Definition (Lamport): A multiprocessor is sequentially consistent if
  - "the result of any execution is the same as if the operations of all the processors were executed in some sequential order,
  - and **the operations of each individual processor appear in this sequence in the order specified by its program**".

- SC is most intuitive model: Represents intuitively expected behavior

Example Revisited: SC

Core C1	Core C2	Notes
S1: STORE x = NEW L1: LOAD r1 = y	S2: STORE y = NEW L2: LOAD r2 = x	/* Init: x = 0, y = 0 */

- Assume: Memory model is **SC**
- No reordering of memory accesses allowed
- What are possible outcomes for the tuple (r1, r2)?
  - (0, NEW) for execution S1, L1, S2, L2
  - (NEW, 0) for execution S2, L2, S1, L1
  - (NEW, NEW) for execution S1, S2, L1, L2
- (0,0) not possible, because it violates program order requirement of SC

Disadvantages of SC

- SC is not used as memory model in modern architectures
  - Today's processor cores use out-of-order execution (better pipeline utilization)
  - STORE instructions are buffered if memory update takes very long due to a cache miss → Go on with calculation while buffering STORE locally

- Enforced orderings of SC prevents many optimizations → Performance slowdown

- Most expensive: Enforced StoreLoad ordering
  - Prevents reordering Stores after Loads that come later in program order
  - Store before a Load (in program order) must be finished (i.e., visible to all other processors) before the Load operation takes place

Relaxing SC: Total Store Order (TSO)

- **Total Store Order (TSO)** memory model allows StoreLoad reordering
- Remaining enforced orderings
  - LoadLoad
  - LoadStore
  - StoreStore
- x86 memory model follows TSO
- Comparison: TSO and SC
  - The TSO model is **weaker** than the SC model.
  - Or: The SC model is **stronger** than the TSO model.

Example Revisited: TSO

Core C1	Core C2	Notes
S1: STORE x = NEW L1: LOAD r1 = y	S2: STORE y = NEW L2: LOAD r2 = x	/* Init: x = 0, y = 0 */

- Assume: Memory model is **TSO**
- StoreLoad reordering of memory accesses allowed (per core)
- What are possible outcomes for the tuple (r1, r2)?
  - (0, NEW) for execution S1, L1, S2, L2
  - (NEW, 0) for execution S2, L2, S1, L1
  - (NEW, NEW) for execution S1, S2, L1, L2
  - (0, 0) for execution L1, S2, L2, S1

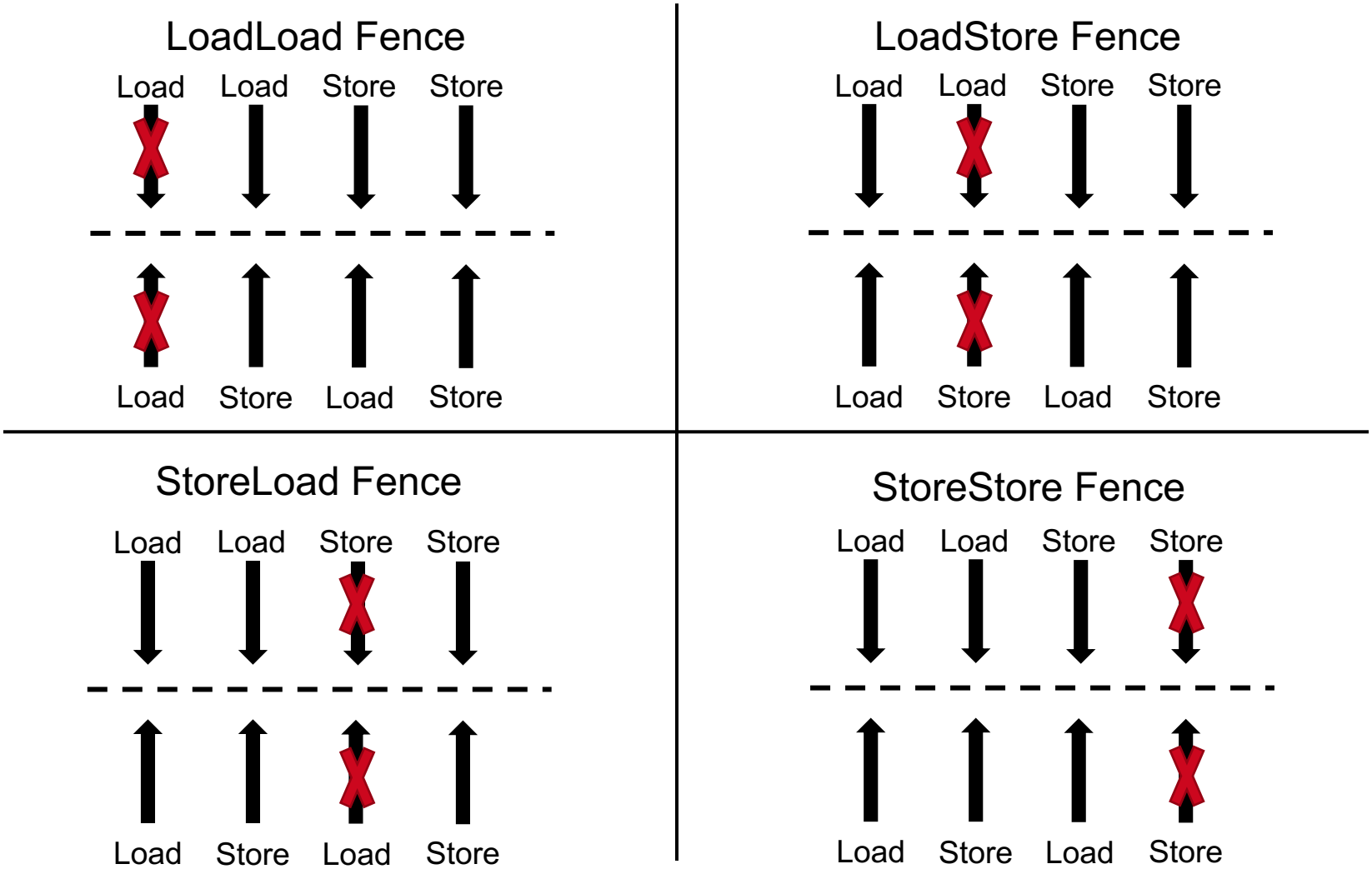
Memory Fences (1)

- How to avoid StoreLoad reordering in TSO explicitly?
- **Memory fence**: Instruction inserted in source code (by programmer or compiler) that explicitly enforces a memory ordering
  - **Full memory fence:** All operations before the fence are finished before all other operations after the fence
  - **StoreLoad fence:** All Store operations before fence are finished before all Load operations after fence
  - **LoadLoad, StoreStore, LoadStore:** similar

Core C1	Core C2	Notes
S1: STORE x = NEW	S2: STORE y = NEW	/* Init: x = 0, y = 0 */
F1: Store_Load_Fence()	F2: Store_Load_Fence()	
L1: LOAD r1 = y	L2: LOAD r2 = x	

- Store-Load fence avoids (0, 0) as a result

Memory Fences (2)



Memory Models Overview

- Hardware memory model: Memory ordering behavior at runtime
- **Software memory model**: Putting another memory model on top of a (typically weaker) hardware memory model
  - “Emulates” a stronger memory model on weaker hardware model by issuing the corresponding memory fences

Source: https://preshing.com/20120930/weak-vs-strong-memory-models/

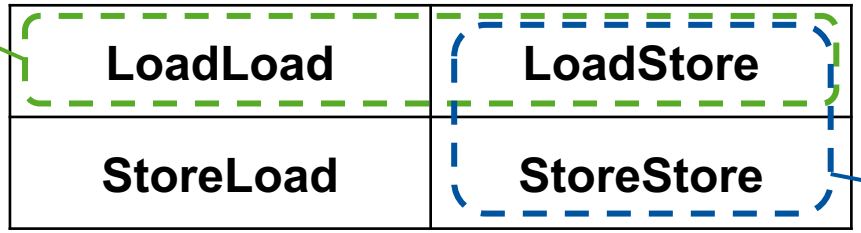
Software Memory Models in C++

- C++ provides different memory orders to guarantee consistency of non-atomic memory accesses around *atomic* operations

`memory_order_`	Fences
`relaxed`	None
`consume`	LoadLoad*, LoadStore*
`acquire`	LoadLoad, LoadStore
`release`	LoadStore, StoreStore
`acq_rel`	LoadLoad, LoadStore, StoreStore
`seq_cst`	All (default order)

* fences only for variables that are data-dependent on that atomic

acquire / consume* semantics



release semantics

Source: https://preshing.com/20120913/acquire-and-release-semantics/

Atomics in C++

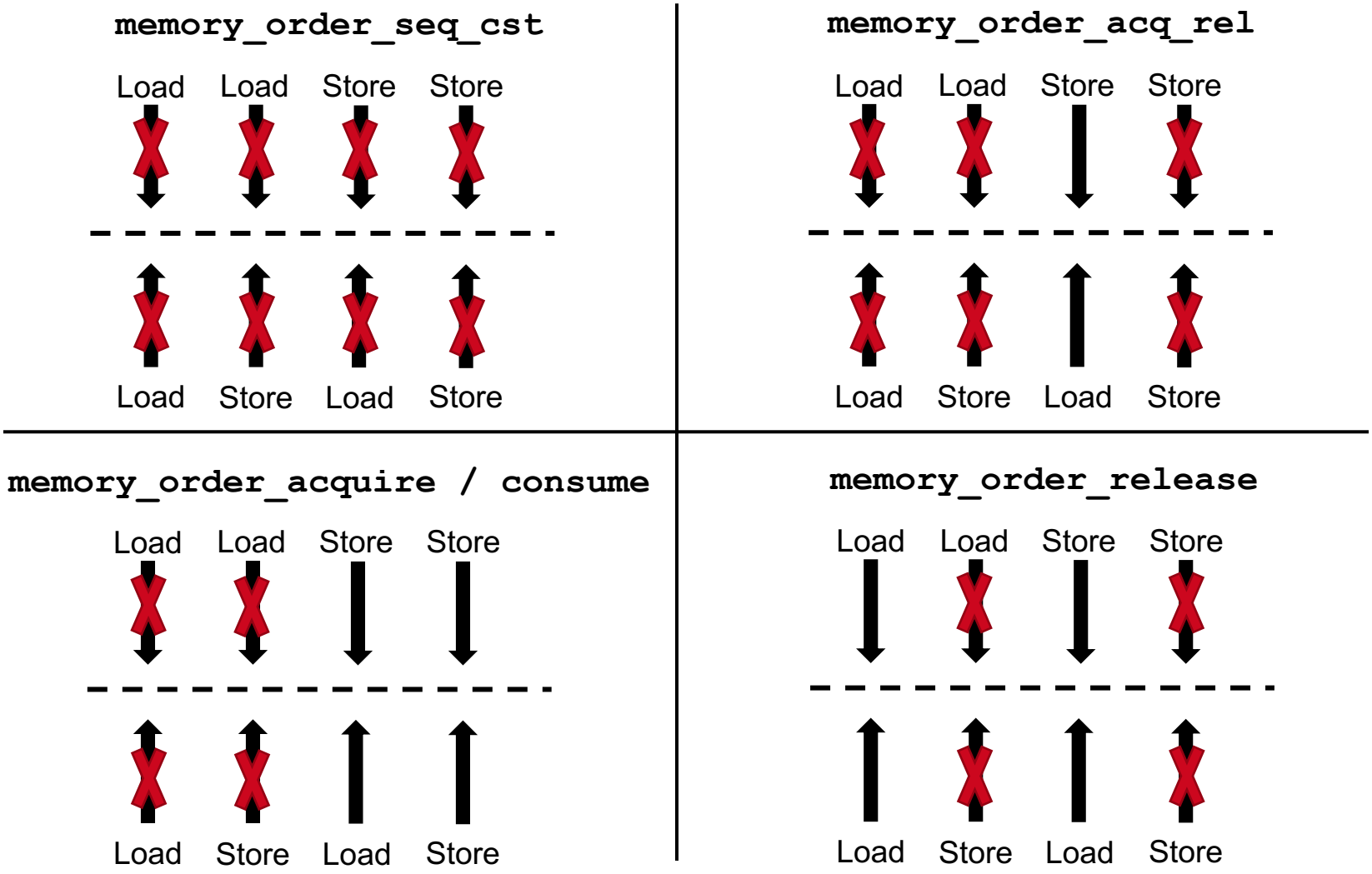
- Atomic operation: Indivisible operation, cannot be observed “half-done” by any thread
- Typically used as synchronization of shared memory accesses between threads, not only just an “atomic” in the common sense
- Standard atomic types in **`<atomic>`** header
- Note: Atomic operations might be lock-based, member function **`is_lock_free()`** can be used to check whether atomic instructions are used (exception: **`std::atomic_flag`** is definitely lock-free)
- **`std::atomic<type>`** declares an atomic variable of the given **`type`**
  - Typical atomic variables: **`atomic<int>`**, **`atomic<bool>`**

Atomic Operations

- Three categories of atomic operations supported with different memory ordering semantics:
  - **Load:** Atomically read from variable (`res = myvar.load()`)
  - **Store:** Atomically write to variable (`myvar.store(val)`)
  - **Read-Modify-Write:** Atomically read *and* write to variable (`res = myvar.exchange(val)`)
- Default memory order for all operations: `memory_order_seq_cst`
- Supported memory orders for different operations

	relaxed	consume	acquire	release	acq_rel	seq_cst
Load	✓	✓	✓			✓
Store	✓			✓		✓
Read-Modify-Write	✓	✓	✓		✓	✓

Memory Fences C++ Atomics



Sequentially Consistent

- Store `data = 1337` of T1 line 5 will be visible by T2 in line 11

```
1   std::atomic<bool> flag(false);
2   int data;
3
4   void thread1() {
5       data = 1337;
6       // full fence (data = 1337 made visible to all threads)
7       flag.store(true, std::memory_order_seq_cst);
8   }
9
10  void thread2() {
11      while (!flag.load(std::memory_order_acquire));
12      // full fence (load of data in assertion cannot be reordered)
13      assert(data == 1337); // will never fail
14  }
15
16  int main() {
17      std::thread t1(thread1), t2(thread2);
18      t1.join(); t2.join();
19  }
```

Acquire / Release

- Remove unnecessary memory fences, no change in semantics

```
std::atomic<bool> flag(false);
int data;

void thread1() {
    data = 1337;
    // StoreStore / LoadStore fence (data = 1337 made visible)
    flag.store(true, std::memory_order_release);
}

void thread2() {
    while (!flag.load(std::memory_order_acquire));
    // LoadLoad / LoadStore fence
    assert(data == 1337); // will never fail
}

int main() {
    std::thread t1(thread1), t2(thread2);
    t1.join(); t2.join();
}
```

Relaxed

- No memory fences, assertion can fail.

```
std::atomic<bool> flag(false);
int data;

void thread1() {
    data = 1337;
    // No memory fence!
    flag.store(true, std::memory_order_relaxed);
}

void thread2() {
    while (!flag.load(std::memory_order_relaxed));
    // No memory fence!
    assert(data == 1337); // can fail
}

int main() {
    std::thread t1(thread1), t2(thread2);
    t1.join(); t2.join();
}
```