Memory organization in multi-core and parallel architectures:

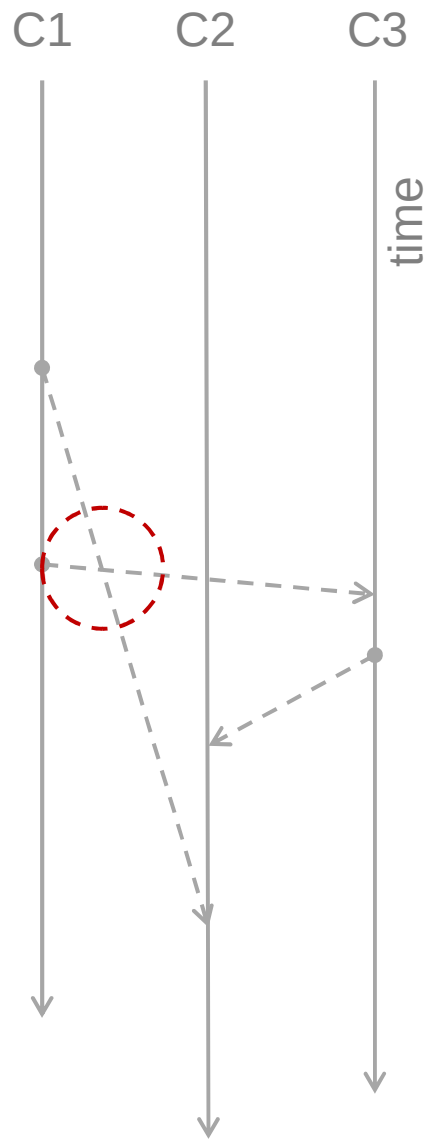
Consistency models

Memory “Consistency”

- When multiple shared-memory processors/cores run concurrently it might be not trivial to define what a “***correct execution***” should be
  - processors/cores run indipendently, each with its own notion of time: what do “time” and “time precedence” actually mean here?
- Consistency provides rules about read and write operations and how they act upon memory
  - concern the loads and stores of *multiple* threads and usually *allow many correct executions* while disallowing many (more) incorrect ones
- many possible legal interleavings of instructions
- Unlike coherence, consistency *is visible* to software
  - however, coherence protocols play an important role in providing consistency

Memory Consistency: basic concepts

- Real-life example of the university registrar
  - (see the book)
  - Consistency model defines whether this behavior is correct (and thus whether a user must take other action to achieve the desired outcome) or incorrect (in which case the system must *preclude these reorderings*).
- Reordering in processors/cores
  - shared memory hardware with out-of-order processor cores, write buffers, prefetching, and multiple cache banks, etc…
  - need to define how programmers know what to expect form hardware, and implementors to know what optimizations they can adopt
- Defining behaviours of parallel systems:
  - multiprocessor are not deterministic
  - illusion of determinism is often created by software with synchronization idioms



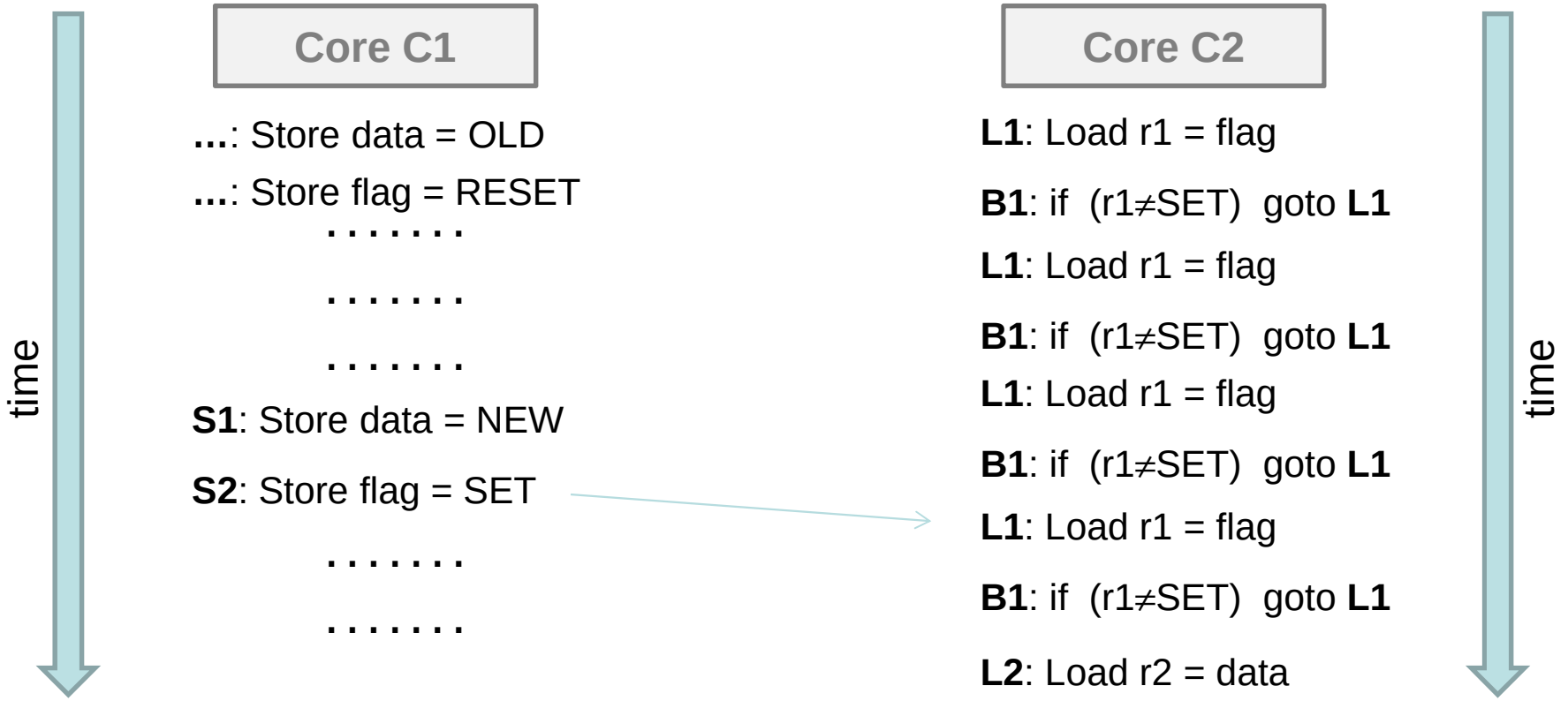
Consistency: a first example

- Look at this example:
  - a low-level synchronization pattern
  - high-level synchronization primitives *might* rely on low-level code looking like this
  - programmer would expect that core C2's register **r2** should get the value **NEW**
  - C2 loops over **L1-B1** until core C1 writes to **flag** (store S2)
  - *Better*: it loops over **L1-B1** *until it sees* C1's write to **flag** !!

Core C1	**Core C2**	**Comments**
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

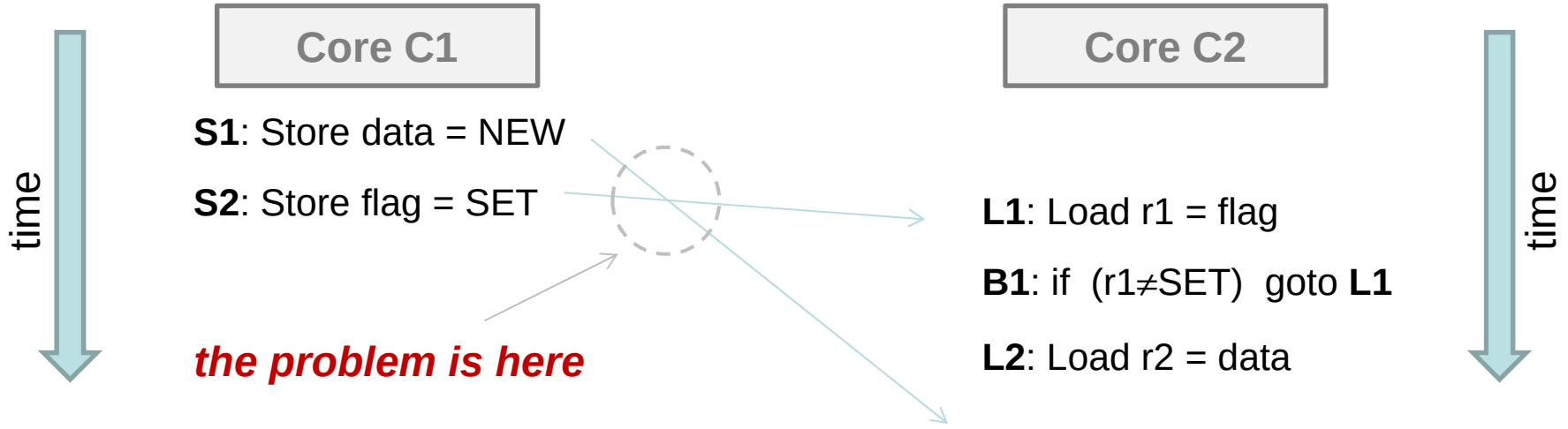
Consistency: a first example

- Look at this example:
  - a *possible* execution (one out of many)
  - intuitively, **r2** should always contain **NEW**



Consistency: a first example

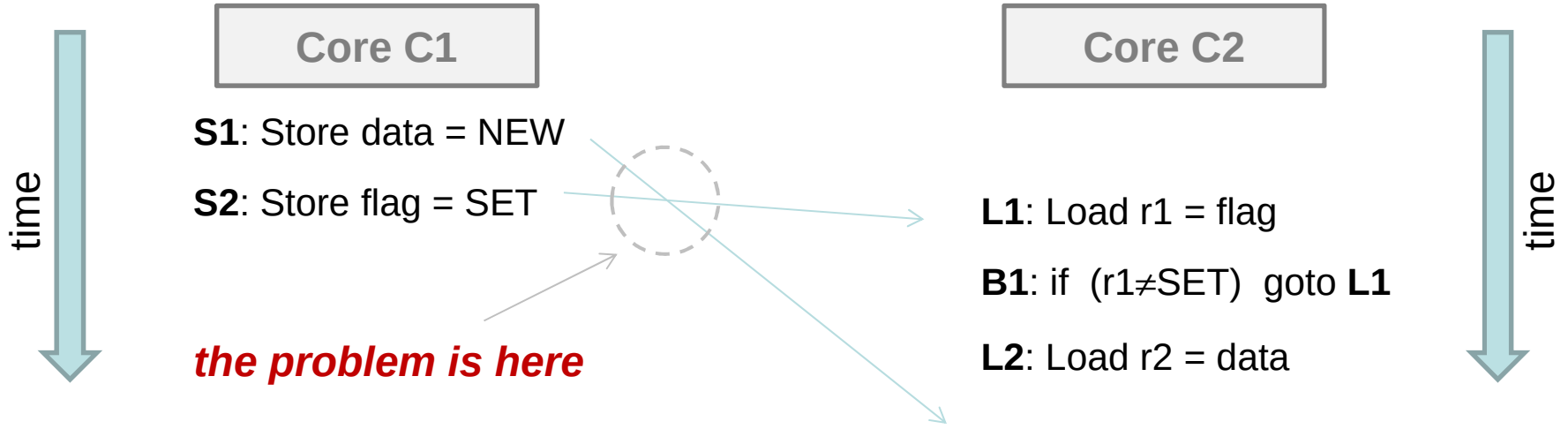
- The particular hardware implementation might reorder C1's stores S1 and S2, e.g. because:
  - Store-store reordering. Two stores may be reordered if a core has a non-FIFO write buffer (e.g. coalescing of non consecutive stores or managing a cache miss)
  - out-of-order execution: e.g. reorder loads L1 and L2 (different addresses)
  - L-S reordering (inverting a Load and a Store in the same queue)
  - locally, these reorderings seem correct to C1 because S1 and S2 access *different* addresses



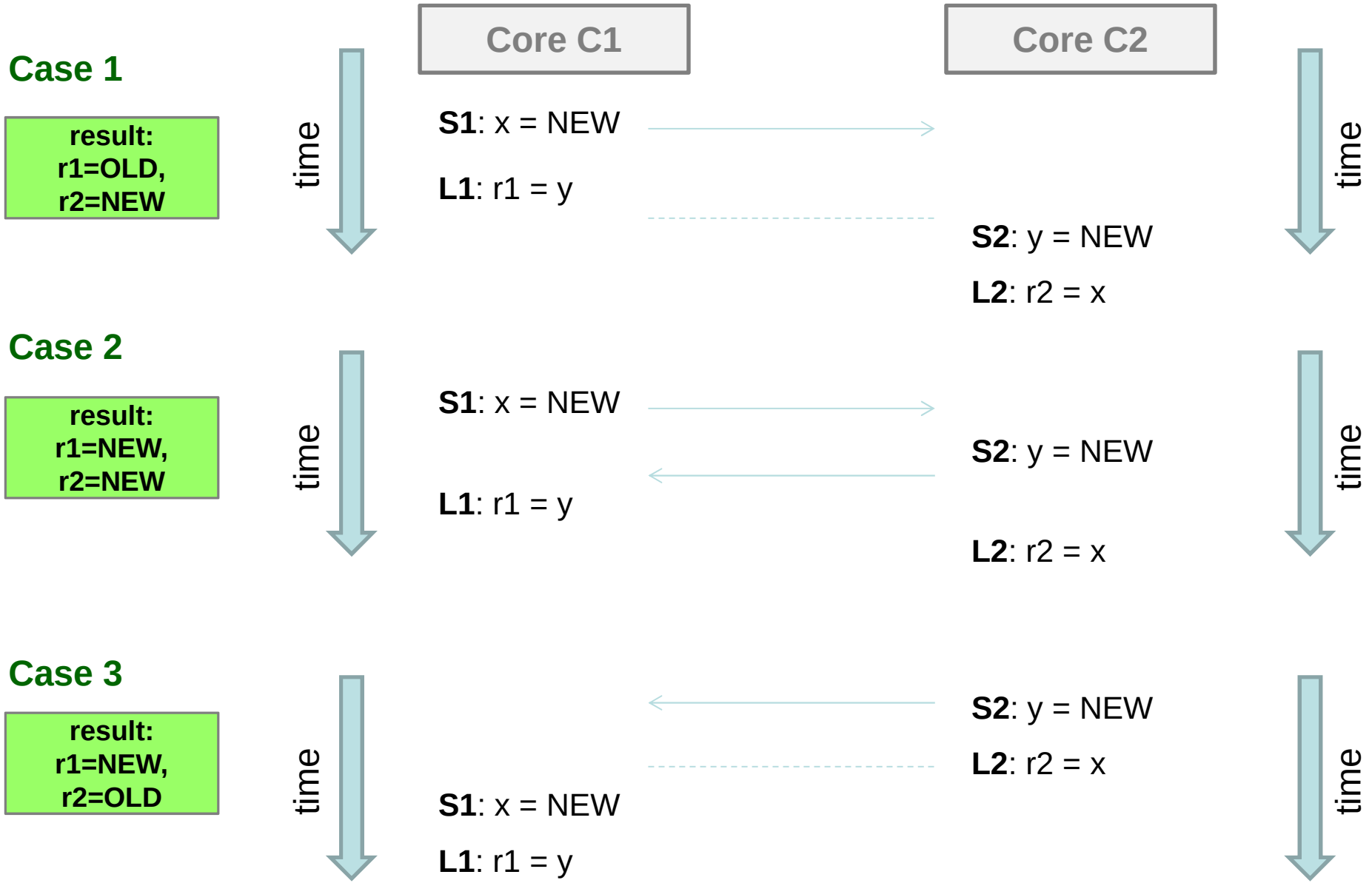
Consistency: a first example

- Look now at this example
- Note: this is not due to coherence problems!

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/

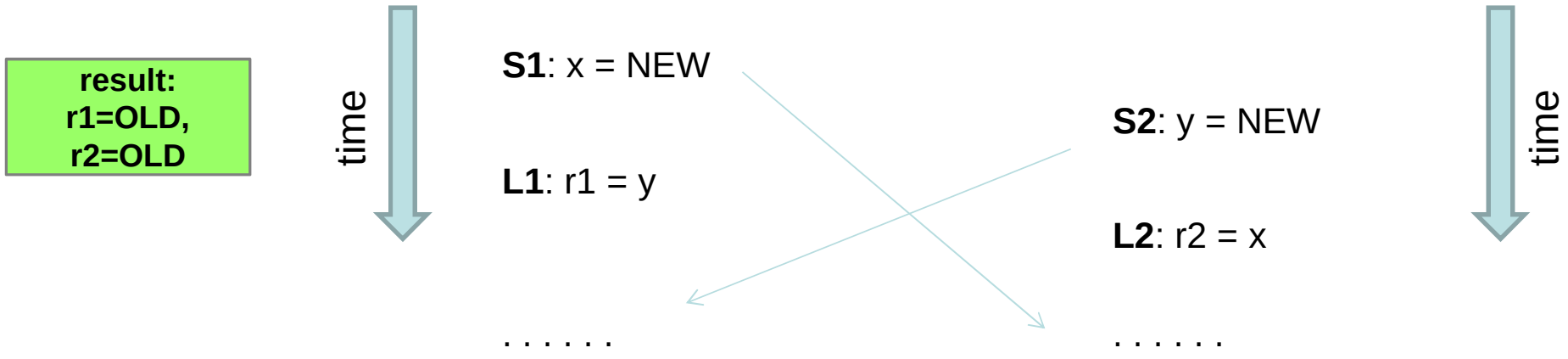


Three *correct* executions



An “incorrect” execution

- It’s actually incorrect under the *Sequential Consistency* (SC) model
- most real hardware, e.g., x86 systems, also allows (**r1**, **r2**) = (**0**, **0**)
  - due to first-in–first-out (FIFO) write buffers used to enhance performance
  - separate write FIFOs can cause stores to take effect at a later moment
- This behaviour *might however be considered correct* under a different consistency model, e.g. TSO
  - Correctness of behaviors is exactly what a Consistency model specifies
  - affects: (a) what behaviors programmers can expect and (b) what optimizations system implementors may use



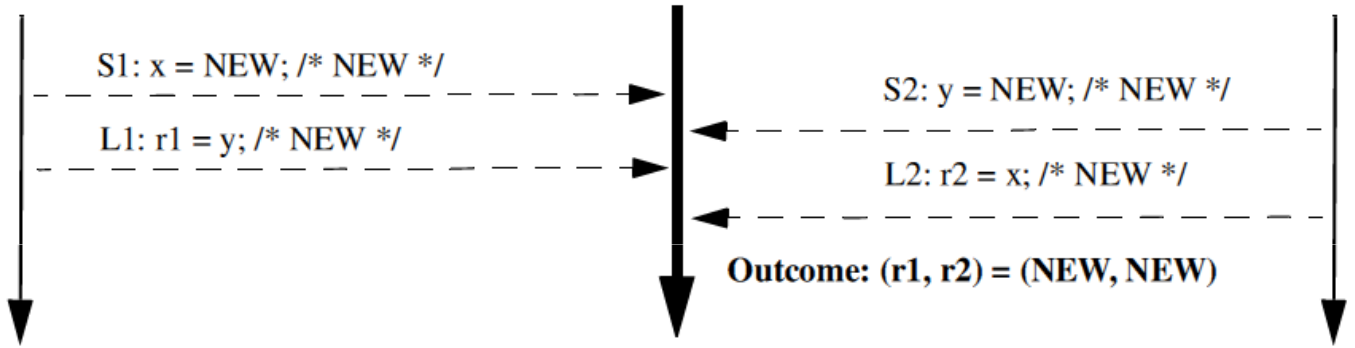
Definition of Sequential Consistency (SC)

- First intuition and definition due to Lamport:
  - **SC** holds if: "the result of *any execution* is the same as if the operations of all 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*"
  - "*any execution*" means: with any possible interleaving of instructions

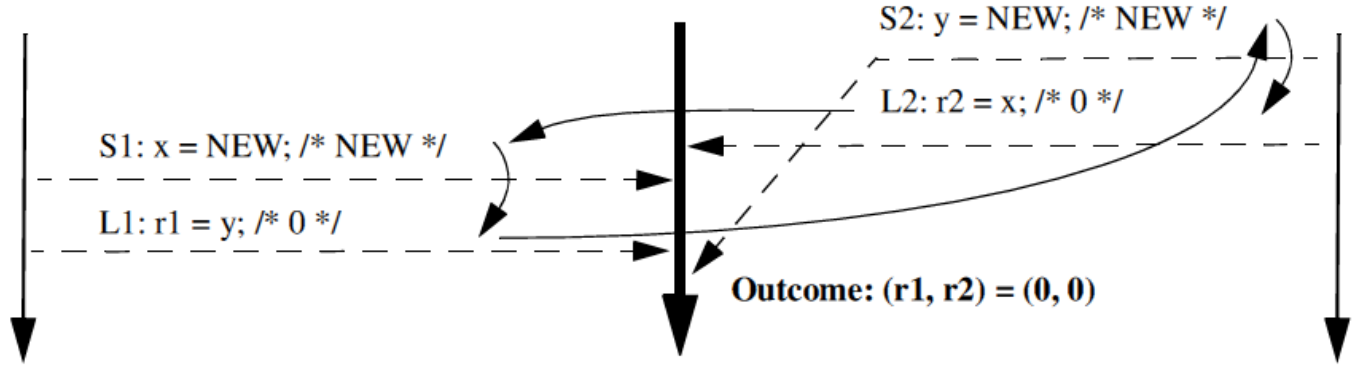
- Notation:
  - **L(a)** and **S(a**) represent a load and a store, resp, to address **a**
  - $<_m$ denotes the *memory order* (order of operations as seen by the memory)
  - $\mathbf{op_1} <_m \mathbf{op_2}$ implies that $\mathbf{op_1}$ precedes $\mathbf{op_2}$ in memory order
  - $<_p$ denotes processor order (order seen by processor ***p***)
  - $\mathbf{op_1} <_p \mathbf{op_2}$ implies that $\mathbf{op_1}$ precedes $\mathbf{op_1}$ in the program on ***p***

Sequential Consistency: example

- Under SC, memory order respects each core's program order. "Respects" means that $\mathbf{op_1} <_p \mathbf{op_2}$ implies $\mathbf{op_1} <_m \mathbf{op_2}$

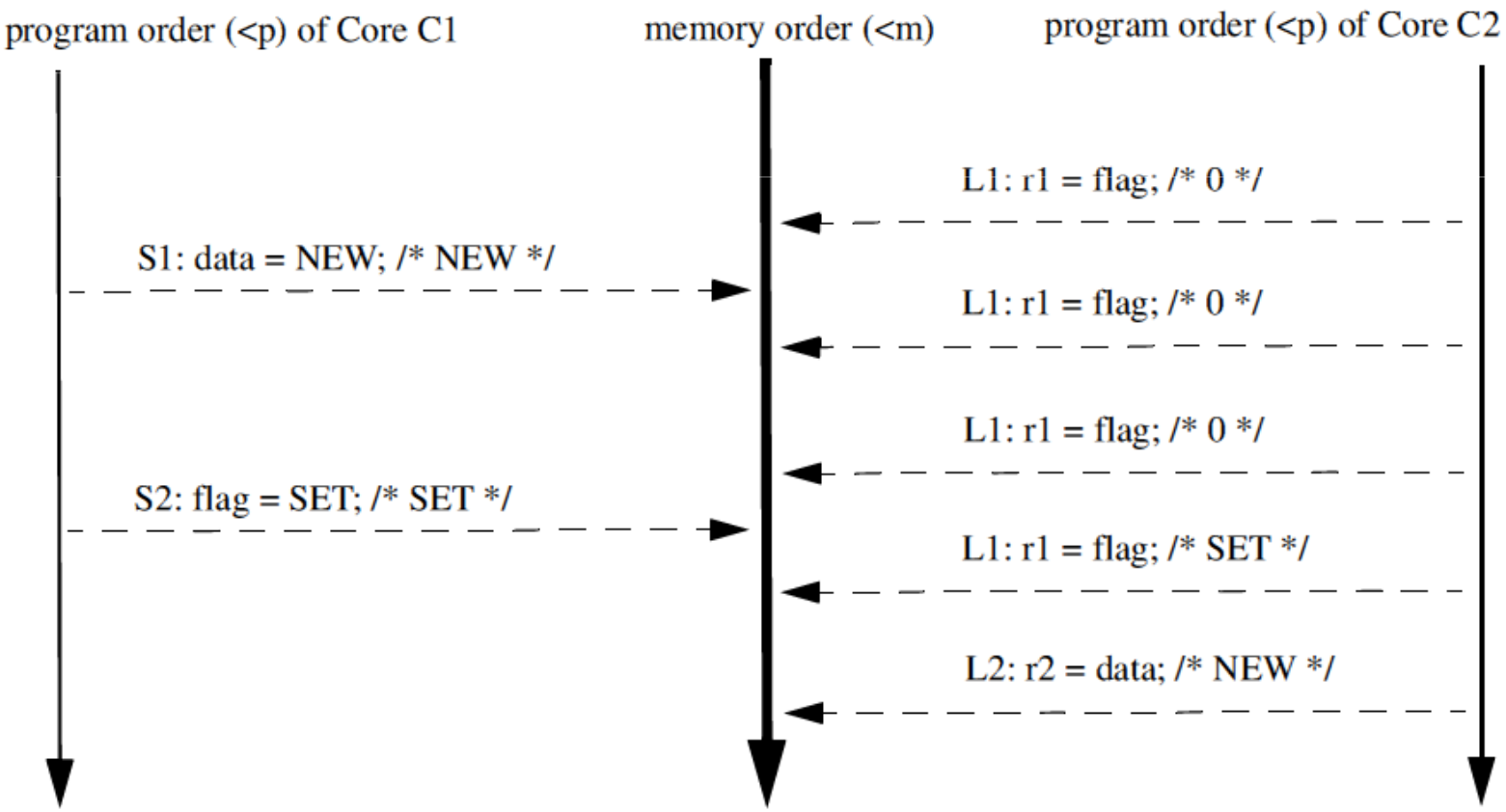


SC property respected



SC property violated:
in program order we have $\mathbf{S_1} <_p \mathbf{L_1}$ *and* $\mathbf{S_2} <_p \mathbf{L_2}$ *while in memory* $\mathbf{L_1} <_m \mathbf{S_2}$ *and* $\mathbf{L_2} <_m \mathbf{S_1}$

Sequential Consistency: example



Formal rules

- Load-Load precedence: $\mathbf{L(a) <_p L(b) \Rightarrow L(a) <_m L(b)}$
- Load-Store precedence: $\mathbf{L(a) <_p S(b) \Rightarrow L(a) <_m S(b)}$
- Store-Store precedence: $\mathbf{S(a) <_p S(b) \Rightarrow S(a) <_m S(b)}$
- Store-Load precedence: $\mathbf{S(a) <_p L(b) \Rightarrow S(a) <_m L(b)}$

- Every load gets its value from the last store before it (in global memory order) to the same address

- Atomic read–modify–write (RMW)

Atomic read–modify–write (RMW)

- “read–modify–write” (RMW) essential for synchronization and used to implement spin-locks and other synchronization primitives
  - e.g., the well-known “test-and-set,” “fetch-and-increment,” and “compare-and-swap
- Implementation:
  - the core to effectively locks the memory system (i.e., it prevents other cores from issuing memory accesses → bad performance)
- Each execution of a test-and-set instruction requires that the load for the test and the store for the set logically appear consecutively in the memory order
  - i.e., no other memory operations for the same or different addresses interpose between them
  - cache block as **M**, no coherence messages or bus locking between Load and Store
  - cache block as **S**, then upgrade, check whether the loaded block was evicted from the cache

Example: R10000 processor

- four-way superscalar RISC, branch prediction, out-of-order execution
  - shared bus and snooping coherence for up to 4 processors
  - dedicated Hubs and directory-based coherence for larger systems (e.g. SGI Origin 2000)
- Loads and stores commit in program order a
- implements SC in cooperation with a cache-coherent memory hierarchy:
  - presents loads and stores in program order to the coherent memory

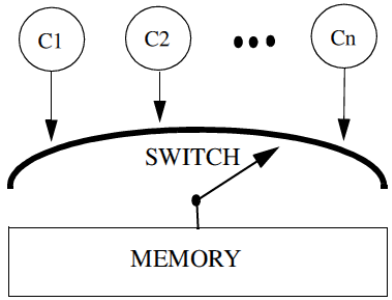
SC formal specification

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

*"**X**" denotes that the two operations must be performed in program order.*

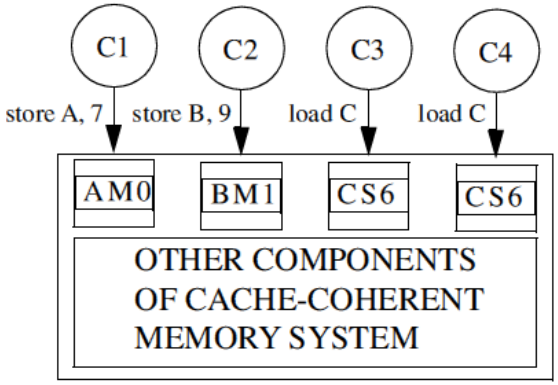
SC “conceptual” implementation models

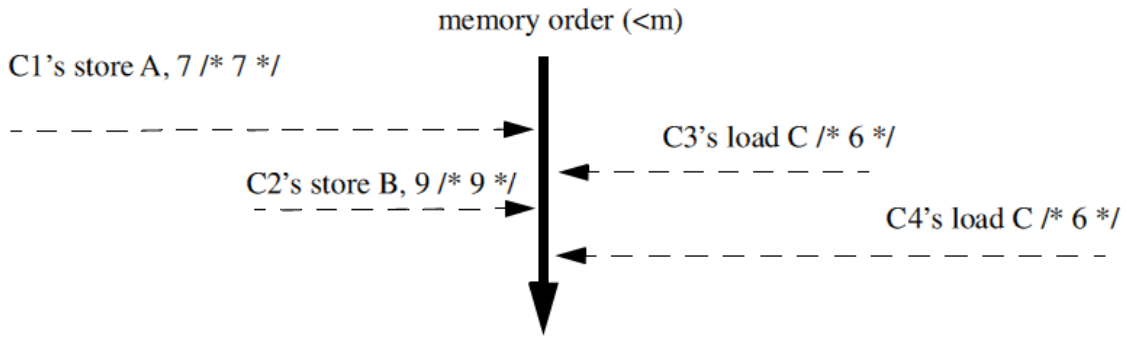
- Multitasking uniprocessor:
  - threads are “time multiplexed” on the same physical core
  - On a context switch, any pending memory operations must be completed before switching
  - SC rules are obeyed by construction by this model
- Multiple processors/cores:
  - memory is multiplexed to cores through a “switch”
  - Cores **can use** optimizations that *do not affect the order* in which it presents memory operations to the “switch”
  - the switch may pick cores by any method, although it should not starve any core having pending operations
  - In practice: The “switch” might physically be a shared bus used for snooping → SC rules are guaranteed by construction (the bus is a single point of serialization), although it’s a performance bottleneck



SC and coherence

- A shared bus used for snooping is an implementation of the switch in the previous abstract implementation model
  - The mechanisms used to serve coherence requests implicitly implement the “selection” performed by the switch
- Cores must still issue coherence requests in program order



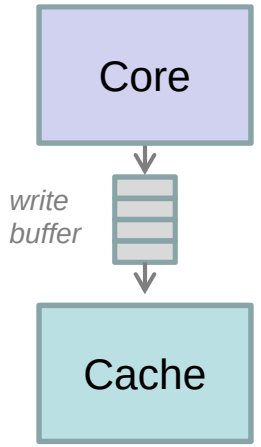


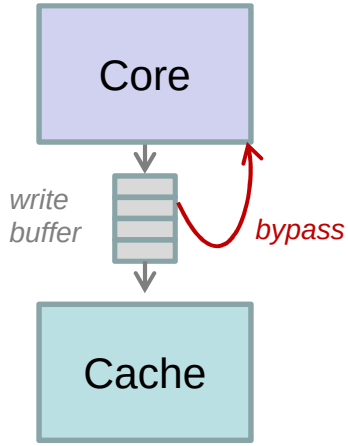
Optimizations compatible with SC

- Non-Binding Prefetching
  - doesn't create any problem for consistency (doesn't modify state):
  - A core may issue coherence requests in any order
- Speculative execution
  - predicted branch wherein subsequent instructions, including Loads and Stores, begin execution, but might require rollback
  - it's compatible with SC, as long as cores do not present their store to the caches until the store is guaranteed to commit
- Cores with dynamically scheduled instructions
  - can easily violate SC (with Load/Stored to different addresses)
  - "Speculate" on SC: predict that reordering isn't visible to other cores
  - Check a speculative load for SC: either verify that the block didn't leave the cache, or repeat the Load to see whether you get the same result
- Multithreading (e.g. SMT)
  - equivalent to distinct "virtual" cores, as long as threads see other threads' activity in memory order (even when sharing the same physical L/S queue)

Total Store Order (TSO) Consistency model

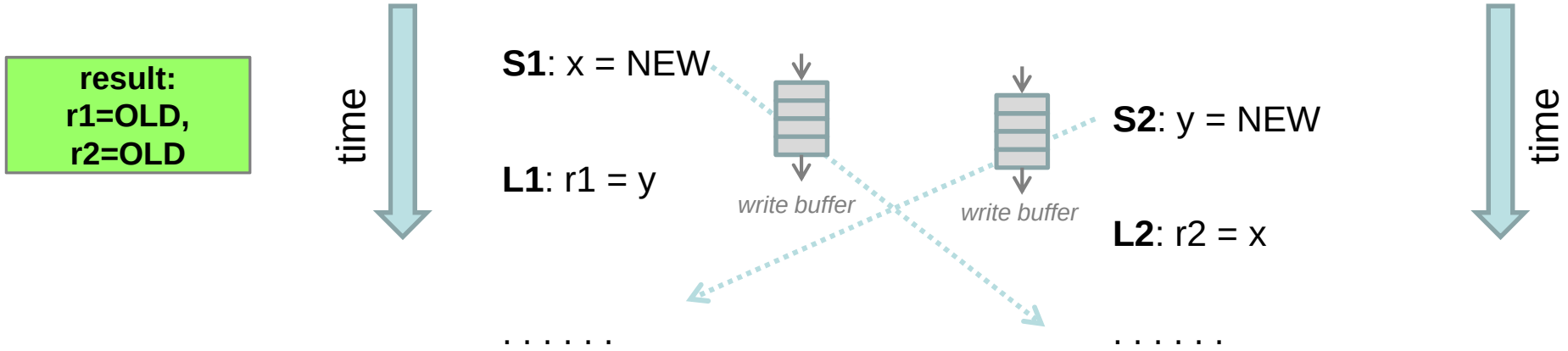
- Sequential Consistency *prevents* many useful implementation optimizations:
  - most notably, write buffers
- **Write buffers** allow cores to hide the latency for serving a store operations
  - particularly, a store miss
- In case of a subsequent load by the same core to the same address:
  - use *bypassing* from the write buffer queue
  - this makes write buffers *architecturally invisible*





How write buffers impact consistency

- The violation of SC seen in a previous example might have been caused by write buffers
  - Write Buffers do not allow implementing SC
- Solutions:
  - disable write buffers (hurts performance)
  - adopt speculative SC (more complexity)
  - give up the complete SC model, adopt a weaker consistency model! → e.g. TSO adopted in Sparc and x86
  - in such a model, the outcome r1=OLD and r2=OLD is admitted!



Formal rules in TSO

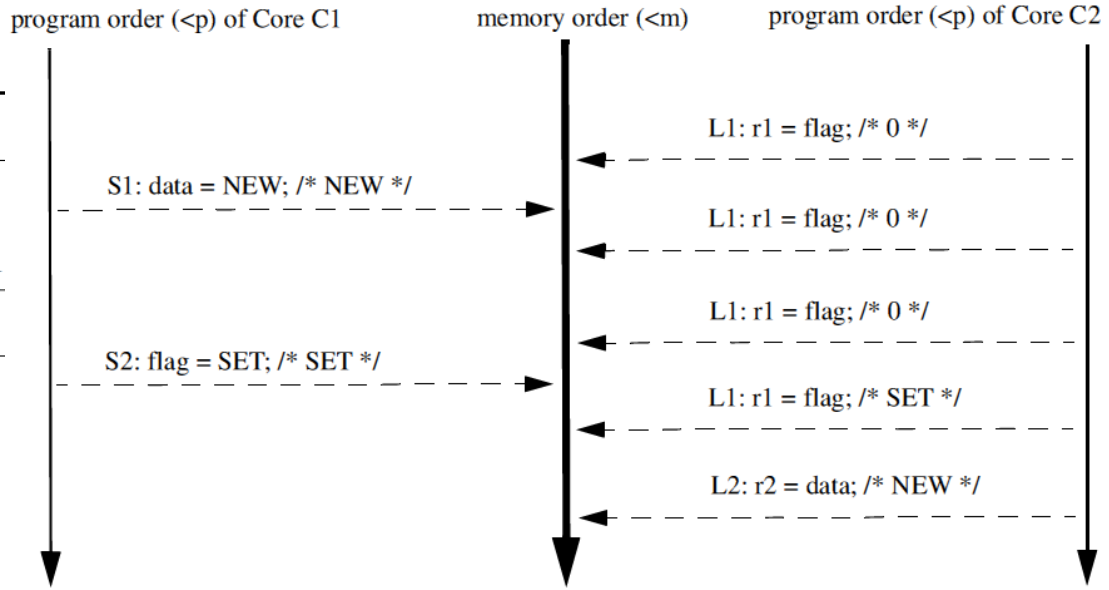
- Load-Load precedence: $\mathbf{L(a) <_p L(b) \Rightarrow L(a) <_m L(b)}$
- Load-Store precedence: $\mathbf{L(a) <_p S(b) \Rightarrow L(a) <_m S(b)}$
- Store-Store precedence: $\mathbf{S(a) <_p S(b) \Rightarrow S(a) <_m S(b)}$
- *Store-Load precedence:* $\mathbf{S(a) <_p L(b) \Rightarrow S(a) <_m L(b)}$
- The fourth rule is omitted in TSO

- Notice:
  - the third rule means that the write buffer must be FIFO (so, for example, coalescing is not supported)
  - preserve store–store order
- In each thread: Every load still gets its value from the last store to the same address
  - this preserves single-thread sequential semantics

TSO is enough most of the time

- Omission of the Store→Load rule doesn't matter for most programs
- In the basic synchronization example given above, TSO allows the same executions as SC
  - it preserves the order of C1's two stores and C2's two (or more) loads
- It's a common parallel programming pattern:
  - C1 reads and writes / C1 writes to a synch variable → C2 reads the synch variable / C2 reads and writes

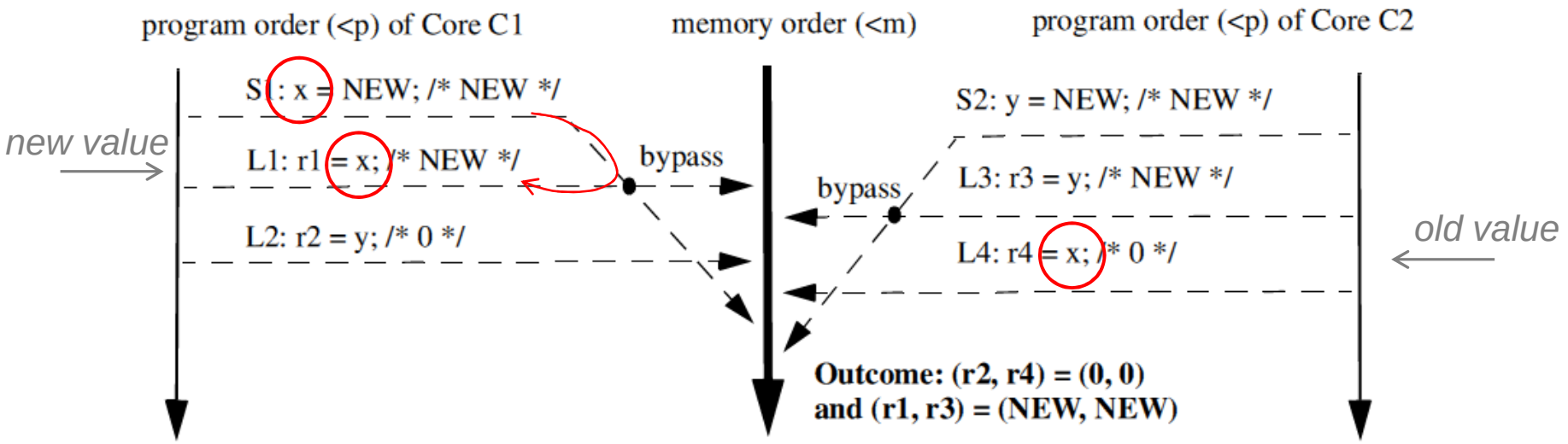
Core C1	Core C2
S1: Store data = NEW;	
S2: Store flag = SET;	L1: Load r1 = flag;
	B1: if (r1 ≠ SET) goto L
	L2: Load r2 = data;



Implications of TSO

- TSO does allow some non-intuitive execution results, which however do not contradict the TSO rules
- See this example, where bypassing comes into play:
  - **r2** and **r4** can get the old values, while **r1** and **r3** get the new ones!

Core C1	Core C2	Comments
S1: x = NEW;	S2: y = NEW;	/* Initially, x = 0 & y = 0*/
L1: r1 = x;	L3: r3 = y;	
L2: r2 = y;	L4: r4 = x;	/* Assume r2 = 0 & r4 = 0 */



FENCE instruction

- A typical instruction provided in multi-processor/multi-core systems
- Executing a FENCE on core Ci ensures that Ci's memory operations before the FENCE (in program order) get placed in memory order before Ci's memory operations after the FENCE
- FENCE defines a precise "point of ordering" in memory
- Note: FENCE ≠ BARRIER(in the parallel programming sense)
  - FENCE doesn't affect order of memory operations at other cores

Core C1	**Core C2**	**Comments**
S1: x = NEW; **FENCE** L1: r1 = y;	S2: y = NEW; **FENCE** L2: r2 = x;	/* Initially, x = 0 & y = 0*/

A previous example, here re-written with the use of **FENCE** instructions which "manually" enforce **Store** → **Load** ordering

Formal rules in TSO

- Load-Fence: **L(a) $<_p$ Fence $\Rightarrow$ L(a) $<_m$ Fence**
- Store-Fence: **S(a) $<_p$ Fence $\Rightarrow$ S(a) $<_m$ Fence**
- Fence-Fence: **Fence $<_p$ Fence $\Rightarrow$ Fence $<_m$ Fence**
- Fence-Load: **Fence $<_p$ L(a) $\Rightarrow$ Fence $<_m$ L(a)**
- Fence-Store: **Fence $<_p$ S(a) $\Rightarrow$ Fence $<_m$ S(a)**

- Because TSO already enforces **L→L**, **L→S**, **S→S**, in TSO we strictly need to specify only two of the above:
  - Store-Fence: **S(a) $<_p$ FENCE $\Rightarrow$ S(a) $<_m$ Fence**
  - Fence-Load: **Fence $<_p$ L(a) $\Rightarrow$ Fence $<_m$ L(a)**

- The **S→L** constraint, when needed, is “manually” enforced by the programmer with a Fence instruction

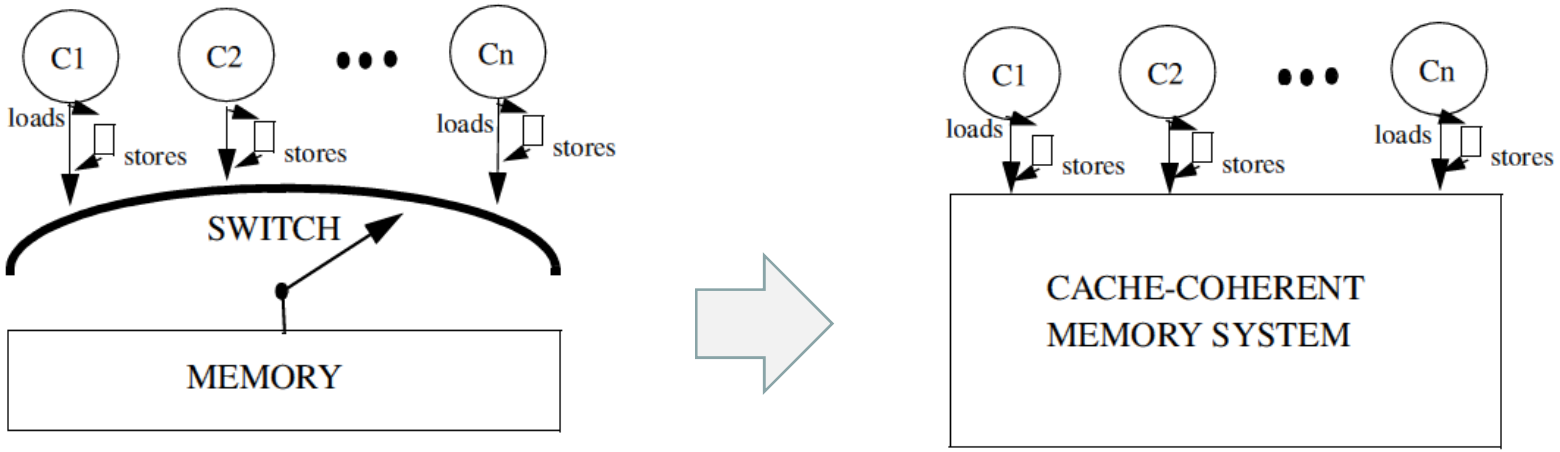
TSO: summary

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	**X**
	Store	**B**	X	X	**X**
	RMW	X	X	X	**X**
	FENCE	**X**	**X**	**X**	**X**

NOTE: **B** *denotes bypassing in case a* **Store→Load** *pair is executed on the same core with the same address*

Conceptual implementation model for TSO

- Similar to SC, simply add FIFOs and enforce TSO rules:
  - Loads/Stores leave cores in program order, Loads either bypass or wait for the switch, the switch picks either a load or the first enqueued store
- Cores could be speculative and/or multithreaded, and use nonbinding prefetches, as in SC
- Most TSO systems take an SC implementation and add write buffers
- *Caveat*: in multithreaded cores (SMT), threads are logically isolated
  - no Load from one thread should bypass from a Store of another thread
  - use write buffer with entries tagged by thread-context identifiers



Implementing Fence and atomic operations

- Implementing atomic operations with TSO:
  - *draining the write buffer* before the Load (R part in RMW), because an early store cannot be delayed after the load, as this would require delaying it also after the Store (W part in RMW), and TSO doesn't allow that
  - acquiring the cache block in read-write mode (for the W part)
  - checking that the cache block wasn't relinquished between R and W parts
  - optimizations are possible that avoid the costly buffer draining
- Implementing Fence with TSO:
  - draining the write buffer, so that all Stores preceding the Fence take effect on memory
  - implementation might be costly, but Fence is TSO are rare because TSO only permits S→L reordering

Relaxed consistency models

- SC and TSO involve strict constraints on implementation
- Fewer ordering constraints can facilitate higher performance
  - allow more hardware and software (compiler and runtime system) optimizations
- Important insight:
  - full ordering constraints might be unnecessary in many circumstances

An example

- The correct behavior of this code requires:
  - S1 → S3 → L1 loads SET → L2
  - S2 → S3 → L1 loads SET → L3
- SC and TSO also require orders S1 → S2 and L2 → L3
  - not needed by the program for correct operation

order not needed here

Core C1	Core C2	Comments
S1: data1 = NEW;		/* Initially, data1 & data2 = 0 & flag ≠ SET */
S2: data2 = NEW;		
S3: flag = SET;	L1: r1 = flag;	/* spin loop: L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: r2 = data1;	
	L3: r3 = data2;	

A further example

- Correct execution requires
  - *all* L1i, *all* S1j → **R1** → **A2** → *all* L2i, *all* S2j
- proper operation does not depend on ordering among many loads and stores
  - unless operations are on the same address, of course

Core C1	Core C2	Comments
A1: acquire(lock)		
/* Begin Critical Section 1 */		
Some loads L1i interleaved with some stores S1j		/* Arbitrary interleaving of L1i's & S1j's */
/* End Critical Section 1 */		
R1: release(lock)		/* Handoff from critical section 1*/
	A2: acquire(lock)	/* To critical section 2*/
	/* Begin Critical Section 2 */	
	Some loads L2i interleaved with some stores S2j	/* Arbitrary interleaving of L2i's & S2j's */
	/* End Critical Section 2 */	
	R2: release(lock)	

Potential optimizations prevented by SC and TSO

- Non-FIFO, coalescing Write Buffers:
  - permit coalescing of writes, i.e., two stores that are *not consecutive* in program order (violates TSO)
- More flexible support for speculation
  - execute loads out of program order before they are ready to be committed
  - SC *can* do speculation, but requires a check when operations are no more speculative (even if mis-speculations are rare)
    - more hardware complexity (and power consumption) in SC
- Temporarily break coherence's *single-writer–multiple-reader* invariant
  - allow some cores to load a new value from a store even if the other cores can still load the old value

Relaxed models: basic insights

- The essential insight behind relaxed model is that the programmer *indicates what ordering is actually needed*
- Relaxed models provide a FENCE instruction so that programmers can indicate when order is needed

- Take the “XC” as an example model (as in the book)
- XC (“eXample Consistency”) model rules:

Load → FENCE Store → FENCE FENCE → FENCE

FENCE → Load FENCE → Store

Load → Load to same address

Load → Store to the same address

Store → Store to the same address

Examples

- S1, S2 → **F1** → S3 → L1 loads SET → **F2** → L2, L3
- F2 is needed because L1, L2, and L3 load from different addresses and might be reordered

Core C1	Core C2	Comments
S1: data1 = NEW;		/* Initially, data1 & data2 = 0 & flag ≠ SET */
S2: data2 = NEW;		
F1: FENCE		
S3: flag = SET;	L1: r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	F2: FENCE	
	L2: r2 = data1;	
	L3: r3 = data2;	

Examples

- FENCEs surround each lock acquire and lock release
  - *all* $L1_i$, *all* $S1_j$ → **F13** → R11 → A21 → **F22** → *all* $L2_i$, *all* $S2_j$
- Notice: not all fences are required here

Core C1	Core C2	Comments
F11: FENCE		
A11: acquire(lock)		
F12: FENCE		
Some loads L1i interleaved with some stores S1j		/* Arbitrary interleaving of L1i's & S1j's */
F13: FENCE		
R11: release(lock)	**F21: FENCE**	/* Handoff from critical section 1*/
F14: FENCE	A21: acquire(lock)	/* To critical section 2*/
	F22: FENCE	
	Some loads L2i interleaved with some stores S2j	/* Arbitrary interleaving of L2i's & S2j's */
	F23: FENCE	
	R22: release(lock)	
	F24: FENCE	

Formal rules in Relaxed Models

- Load-Fence: **$L(a) <_p Fence \Rightarrow L(a) <_m Fence$**
- Store-Fence: **$S(a) <_p FENCE \Rightarrow S(a) <_m Fence$**
- Fence-Fence: **$Fence <_p Fence \Rightarrow Fence <_m Fence$**
- Fence-Load: **$Fence <_p L(a) \Rightarrow Fence <_m L(a)$**
- Fence-Store: **$Fence <_p S(a) \Rightarrow Fence <_m S(a)$**

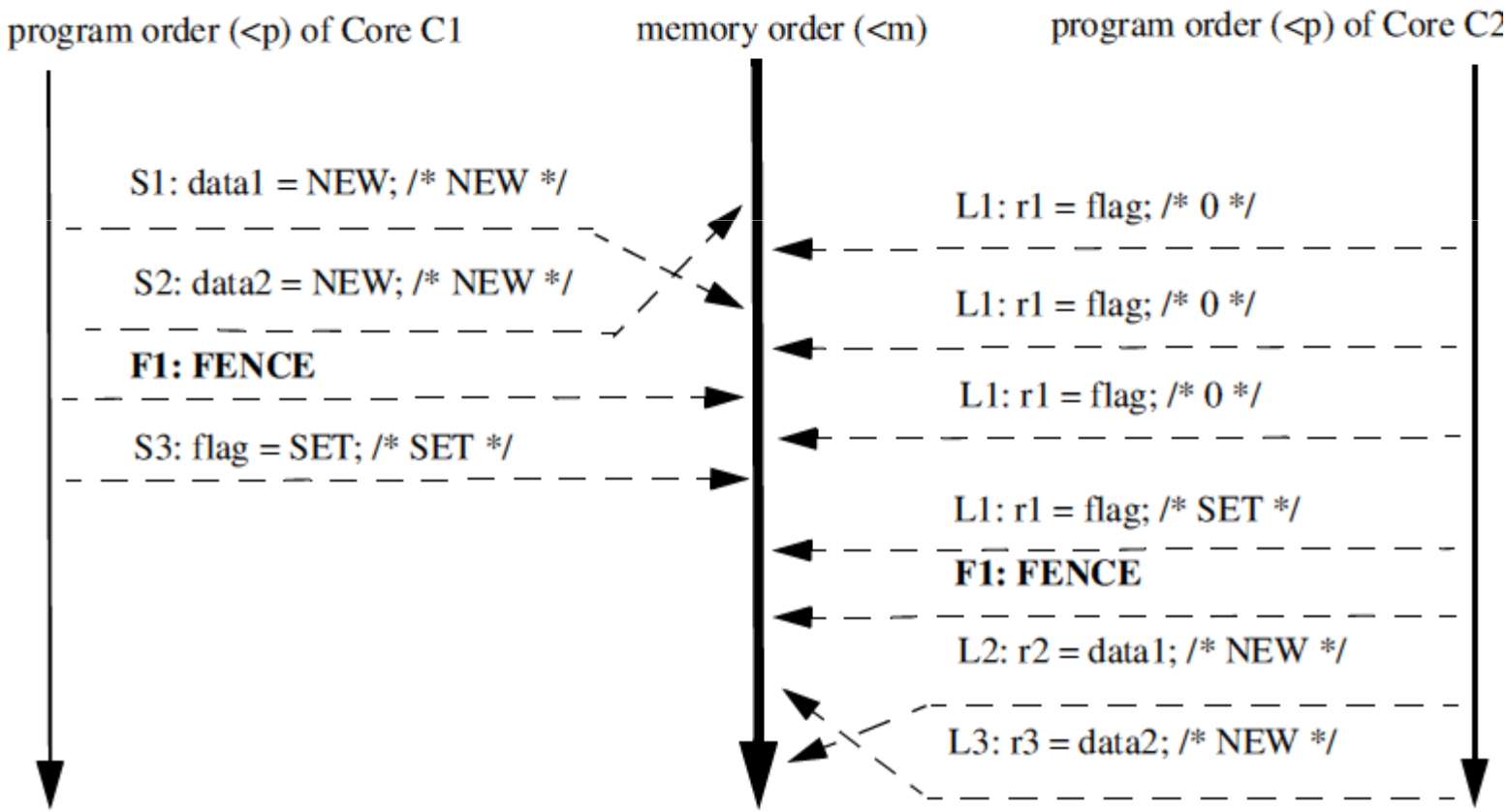
- Loads/stores to the same address in order **$<_m$**:
  - Load-Load (same address): **$L(a) <_p L'(a) \Rightarrow L(a) <_m L'(a)$**
  - Load-Store (same address): **$L(a) <_p S(a) \Rightarrow L(a) <_m S(a)$**
  - Store-Store (same address): **$S(a) <_p S(a) \Rightarrow S(a) <_m S(a)$**

- Unsupported ordering constraints are “manually” enforced by the programmer with suitable Fence instructions

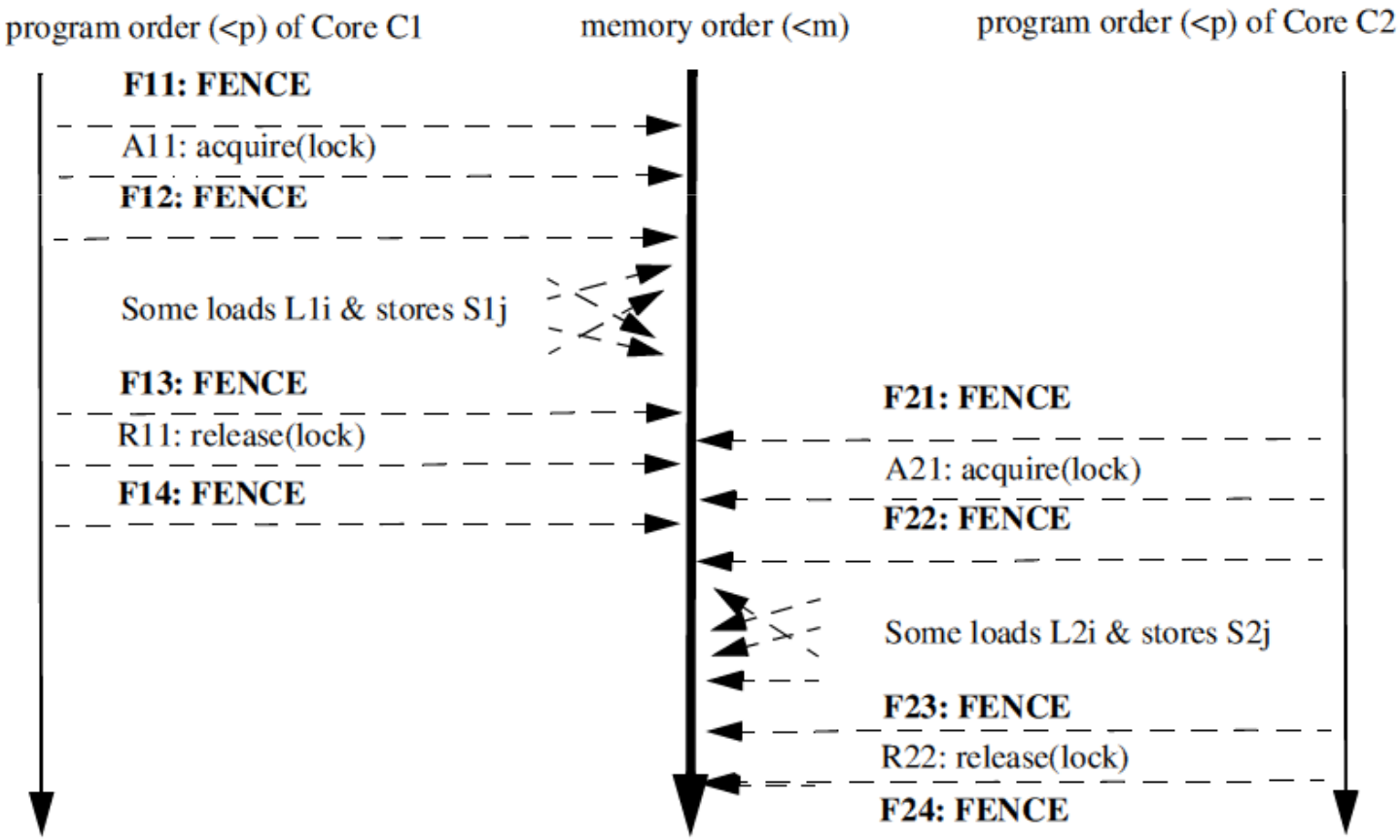
Formal rules in Relaxed Models

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	FENCE	X	X	X	X

Examples

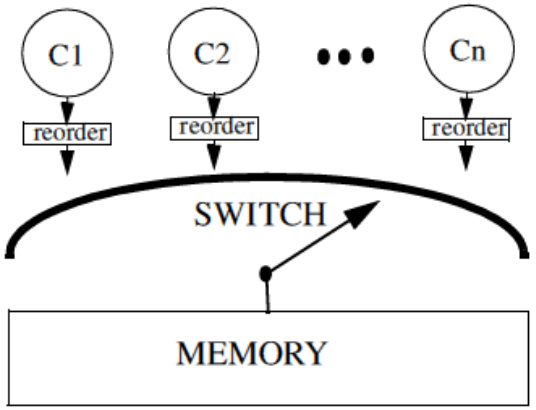

program order (<p) of Core C1
memory order (<m)
program order (<p) of Core C2
S1: data1 = NEW; /* NEW */
S2: data2 = NEW; /* NEW */
F1: FENCE
S3: flag = SET; /* SET */
L1: r1 = flag; /* 0 */
L1: r1 = flag; /* 0 */
L1: r1 = flag; /* 0 */
L1: r1 = flag; /* SET */
F1: FENCE
L2: r2 = data1; /* NEW */
L3: r3 = data2; /* NEW */

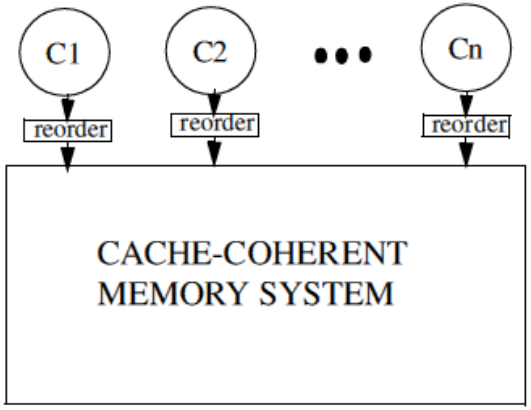
Examples


program order (<p) of Core C1
memory order (<m)
program order (<p) of Core C2
F11: FENCE
A11: acquire(lock)
F12: FENCE
Some loads L1i & stores S1j
F13: FENCE
R11: release(lock)
F14: FENCE
F21: FENCE
A21: acquire(lock)
F22: FENCE
Some loads L2i & stores S2j
F23: FENCE
R22: release(lock)
F24: FENCE

Implementing Relaxed Models

- General conceptual model:
  - separate the reordering of core operations from coherence (as in TSO)
- Each core is separated from memory by a more general *reorder unit*
  - queues operations and places them either in program order or reordered;
  - must obey rules for **(1)** FENCEs, **(2)** L/S to same address, **(3)** bypassing:
  - **(1)** Cannot reorder: Load→FENCE, Store→FENCE, FENCE→FENCE, FENCE→Load, or FENCE→Store
  - **(2)** Cannot reorder: Load→Load, Load→Store, Store→Store to the same address
  - **(3)** loads immediately see updates due to their own stores





Implementing atomic instructions

- As in TSO, when executing an atomic instruction the core:
  - drains the write buffer, gets the block with read–write coherence permissions, and then performs the load/store operation pair
  - cache controller must not evict the block between the load and store
- Sacrifices some performance:
  - draining the write buffer is in fact not required
- Important difference between XC and TSO is how atomic RMWs are used to achieve synchronization:
  - a lock acquire must be followed by a FENCE
  - a lock release must be preceded by a FENCE

Code	**TSO**	**XC**
acquire lock	RMW: test-and-set L /* read L, write L=1 if L==1, goto RMW */ if lock held, try again	RMW: test-and-set L /* read L, write L=1 if L==1, goto RMW */ if lock held, try again **FENCE** ⟵
critical section	loads and stores	loads and stores
release lock	store L=0	**FENCE** ⟵ store L=0

Implementing Fences with relaxed models

- FENCEs must order: "*all* X_i" $<_m$ FENCE $<_m$ "*all* Y_i"
- Three approaches for implementation:
  - implement complete SC if required, and treat all FENCEs as no-operations
  - "FENCE as drain" method: simply wait for all memory operations X_i to perform, consider the FENCE done, and then begin memory operations Y_i)
  - enforce X_i $<_m$ FENCE $<_m$ Y_i, without draining: much more complex to implement
- In all cases, a FENCE implementation must rely on a method to know when each operation X_i is done

Relaxed consistency + Data Race Free code

- Data race:
  - two threads access the same memory location, at least one is a write, and there's no intervening synchronization
- A pair of "conflicting" data operations $D_i <_m D_j$ are *not a data race* if and only if there exists a pair of transitively "conflicting" synchronization operations S_i and S_j such that $D_i <_m S_i <_m S_j <_m D_j$
  - "conflicting" here essentially means interfering with each other
- "SC for DRF (Data Race Free)" programs:
  - ensures that all executions respect the full SC constraints *as long as* programs are free of data races (programmer's care)
  - For example, with FENCEs around synchronization operations, XC supports "SC for DRF" programs
  - "SC for DRF" principles serve as a foundation for Java and C++ high-level language (HLL) memory models

Relaxed consistency + Data Race Free code

- An example with a data-race:

Core C1	Core C2	Comments
F11: FENCE		/* Initially, data1 & data2 = 0 */
A11: acquire(lock)		
F12: FENCE		
S1: data1 = NEW;	L1: r2 = data2;	
S2: data2 = NEW;	L2: r1 = data1;	/* Four Possible Outcomes under XC:
F13: FENCE		(r1, r2) =
R11: release(lock)		(0, 0), (0, NEW), (NEW, 0), or (NEW, NEW)
F14: FENCE		But has a Data Race */

Relaxed consistency + Data Race Free code

- An example with no data-race
  - Outcomes are the same as they would be allowed under SC

Core C1	Core C2	Comments
F11: FENCE		/* Initially, data1 & data2 = 0 */
A11: acquire(lock)		
F12: FENCE		
S1: data1 = NEW;		
S2: data2 = NEW;		
F13: FENCE		
R11: release(lock)	**F21: FENCE**	
F14: FENCE	A21: acquire(lock)	
	F22: FENCE	
	L1: r2 = data2;	
	L2: r1 = data1;	/* Two Possible Outcomes under XC:
	F23: FENCE	(r1, r2) =
	R22: release(lock)	(0, 0) or (NEW, NEW)
	F24: FENCE	Same as with SC */