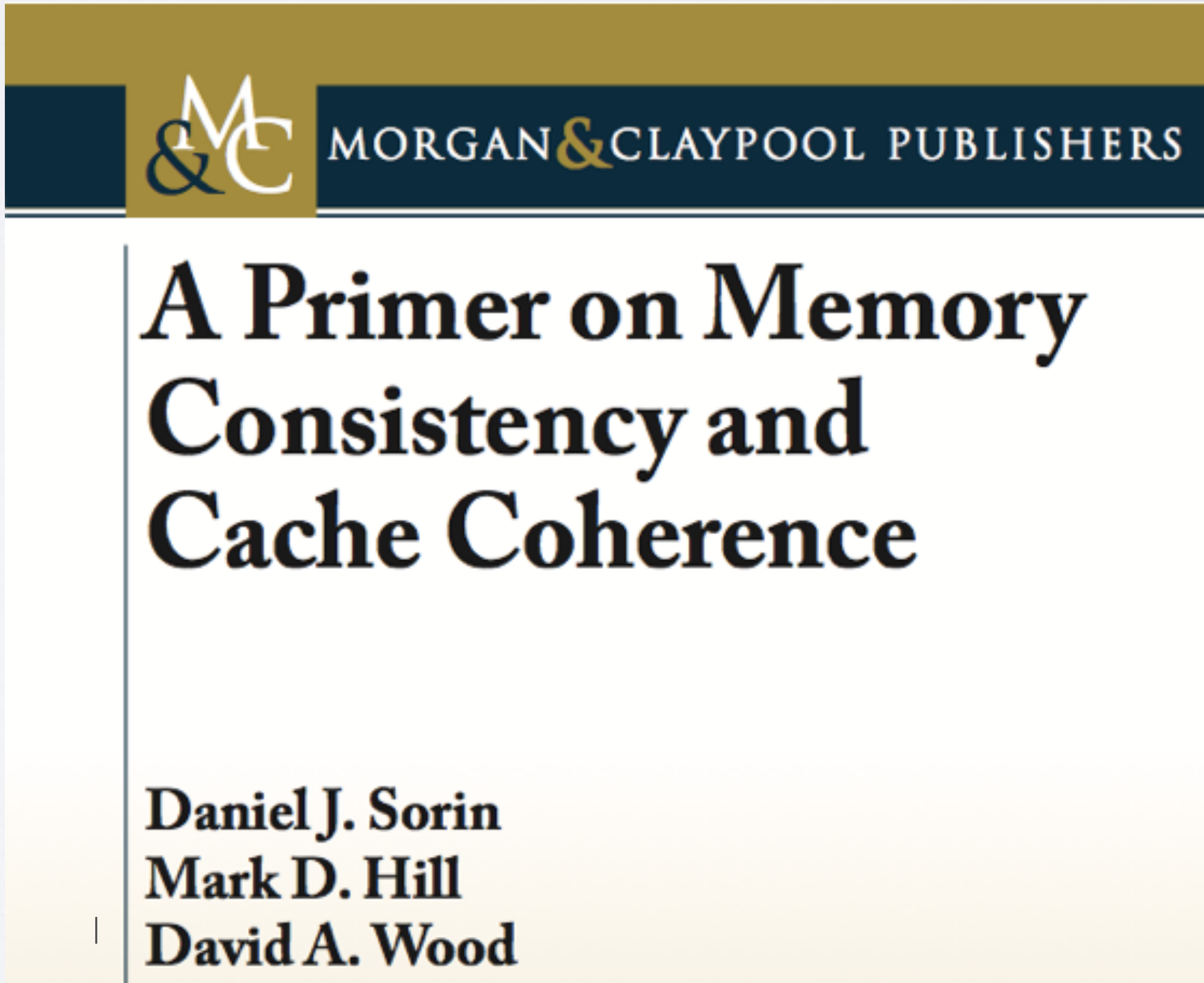
RELAXED CONSISTENCY



MORGAN & CLAYPOOL PUBLISHERS

A Primer on Memory Consistency and Cache Coherence

Daniel J. Sorin
Mark D. Hill
David A. Wood

RELAXED CONSISTENCY

- "Relaxed Consistency" is a catch-all term for any MCM weaker than TSO

- GPUs have relaxed consistency (probably)

XC AXIOMS

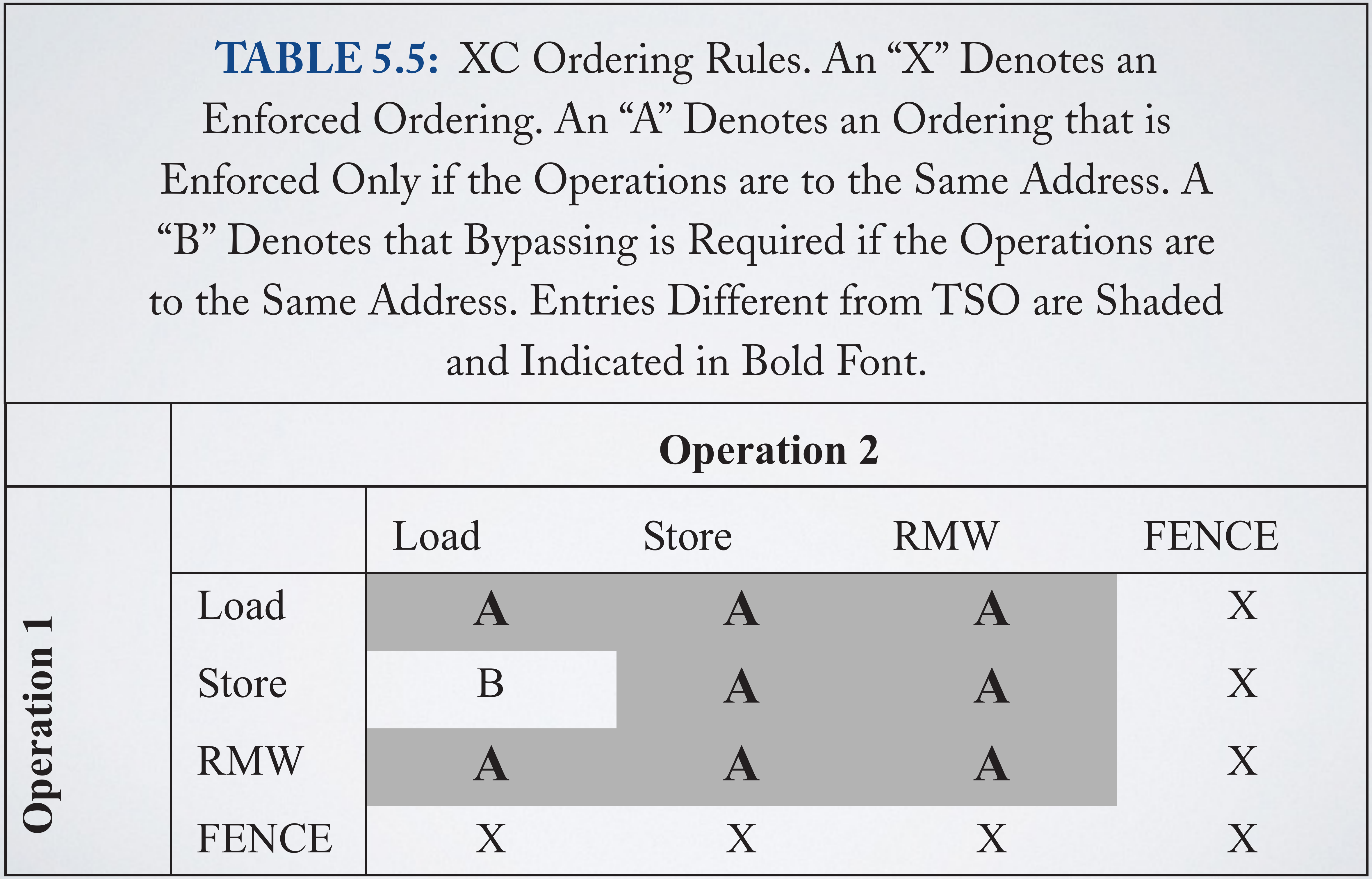
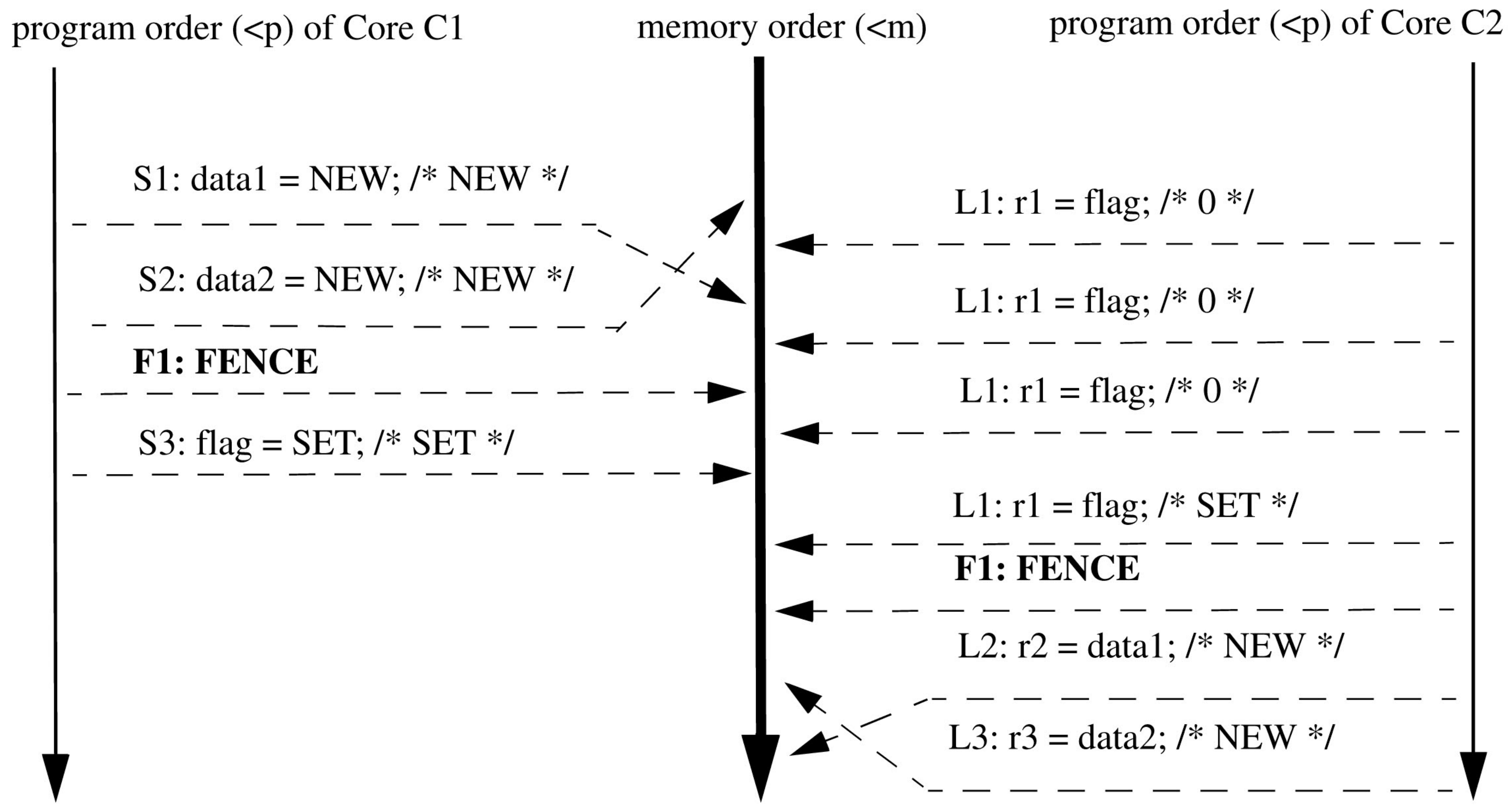


TABLE 5.5: XC Ordering Rules. An "X" Denotes an Enforced Ordering. An "A" Denotes an Ordering that is Enforced Only if the Operations are to the Same Address. A "B" Denotes that Bypassing is Required if the Operations are to the Same Address. Entries Different from TSO are Shaded and Indicated in Bold Font.

		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

XC EXAMPLE


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 */

(a) An XC Execution

XC LOCKS NEED FENCES

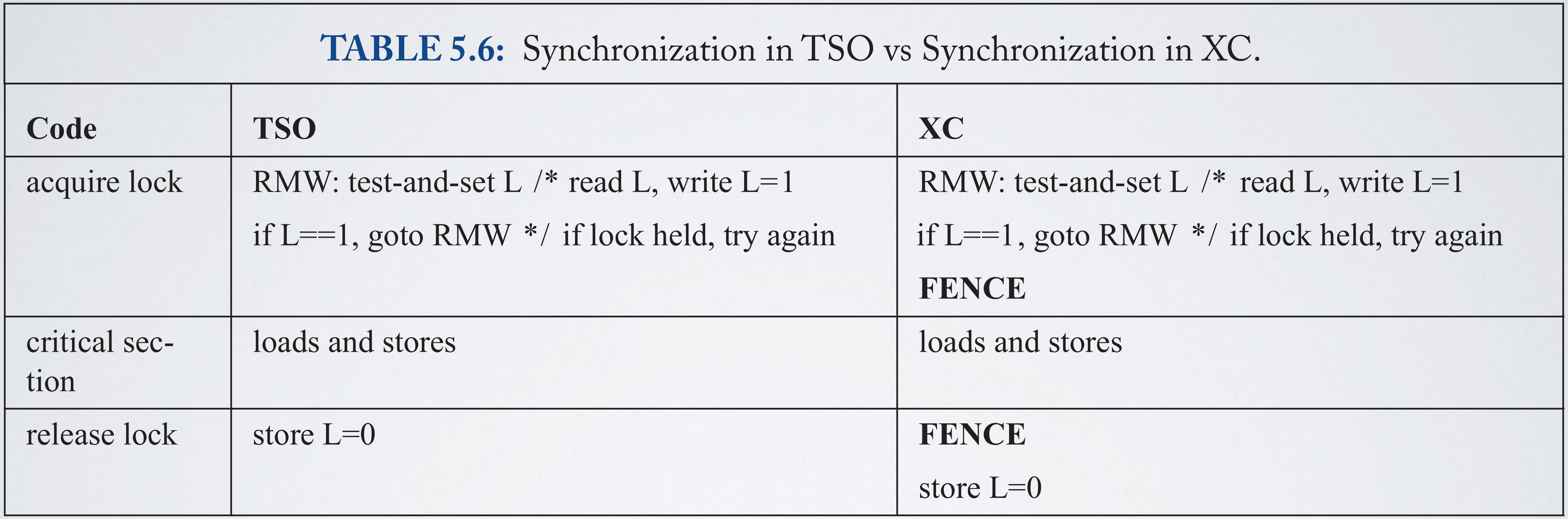


TABLE 5.6: Synchronization in TSO vs Synchronization in XC.

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 sec- tion	loads and stores	loads and stores
release lock	store L=0	**FENCE** store L=0

WHY BOTHER WITH XC?

- coalescing store buffers
- more freedom for OoO execution
- GPUs are similar to XC

DRF => SC

- the **great compromise** between architects/compiler writers and programmers
- definition of a data race:
  - two accesses to the same memory location from different threads
  - at least one access is a store
  - accesses are "unsynchronized"
- Data-race-free **programs** are guaranteed to always have SC **executions**
  - this guarantee is probably* provided by all consistency models

WHAT COUNTS AS SYNCHRONIZATION?

- sync == specially-tagged memory operations
  - usually a pair of insns: load/store/RMW + fence
  - in most ISAs RMW operations come with a fence automatically

WRITE ATOMICITY

- write atomicity := a core's store is logically seen by all **other** cores **at once**
- "Write atomicity allows a core to see the value of its own store before it is seen by other cores, as required by XC, causing some to consider '*write atomicity*' to be a poor name." - *APMCCC*, p. 69

CAUSALITY

TABLE 5.9: Causality: If I See a Store and Tell You About It, Must You See It Too?

Core C1	**Core C2**	**Core C3**
S1: data1 = NEW;		/* Initially, data1 & data2 = 0 */
	L1: r1 = data1;	
	B1: if (r1 ≠ NEW) goto L1;	
	F1: FENCE	
	S2: data2 = NEW;	
		L2: r2 = data2;
		B2: if (r2 ≠ NEW) goto L2;
		F2: FENCE
		L3: r3 = data1; /* r3==NEW? */

INDEPENDENT READS, INDEPENDENT WRITES

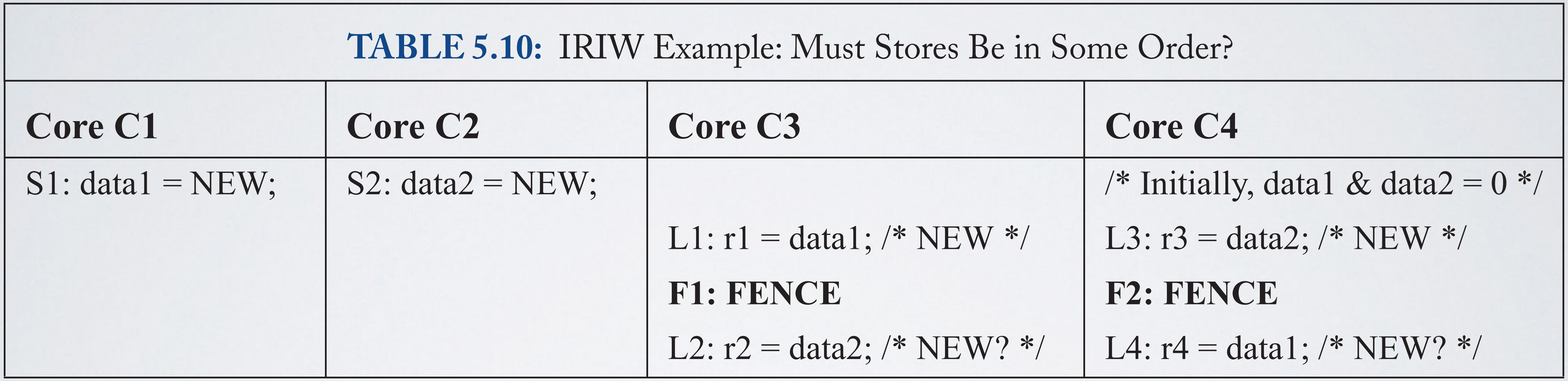


TABLE 5.10: IRIW Example: Must Stores Be in Some Order?

Core C1	**Core C2**	**Core C3**	**Core C4**
S1: data1 = NEW;	S2: data2 = NEW;		/* Initially, data1 & data2 = 0 */
		L1: r1 = data1; /* NEW */	L3: r3 = data2; /* NEW */
		F1: FENCE	**F2: FENCE**
		L2: r2 = data2; /* NEW? */	L4: r4 = data1; /* NEW? */

IBM POWER FENCES

- SYNC or HWSYNC ("HW" means "heavy weight") orders all accesses X before all accesses Y and is cumulative.

- LWSYNC ("LW" means "light weight") orders loads in X before loads in Y, orders loads in X before stores in Y, and orders stores in X before stores in Y. LWSYNC is cumulative. *Note that LWSYNC does not order stores in X before loads in Y.*

IBM POWER NON-FENCE ORDERING

- “Power orders accesses in some cases even without FENCEs. For example, if load L1 obtains a value used to calculate an effective address of a subsequent load L2, then Power orders load L1 before load L2.” - *APMCCC*, p. 72

- How could it not????

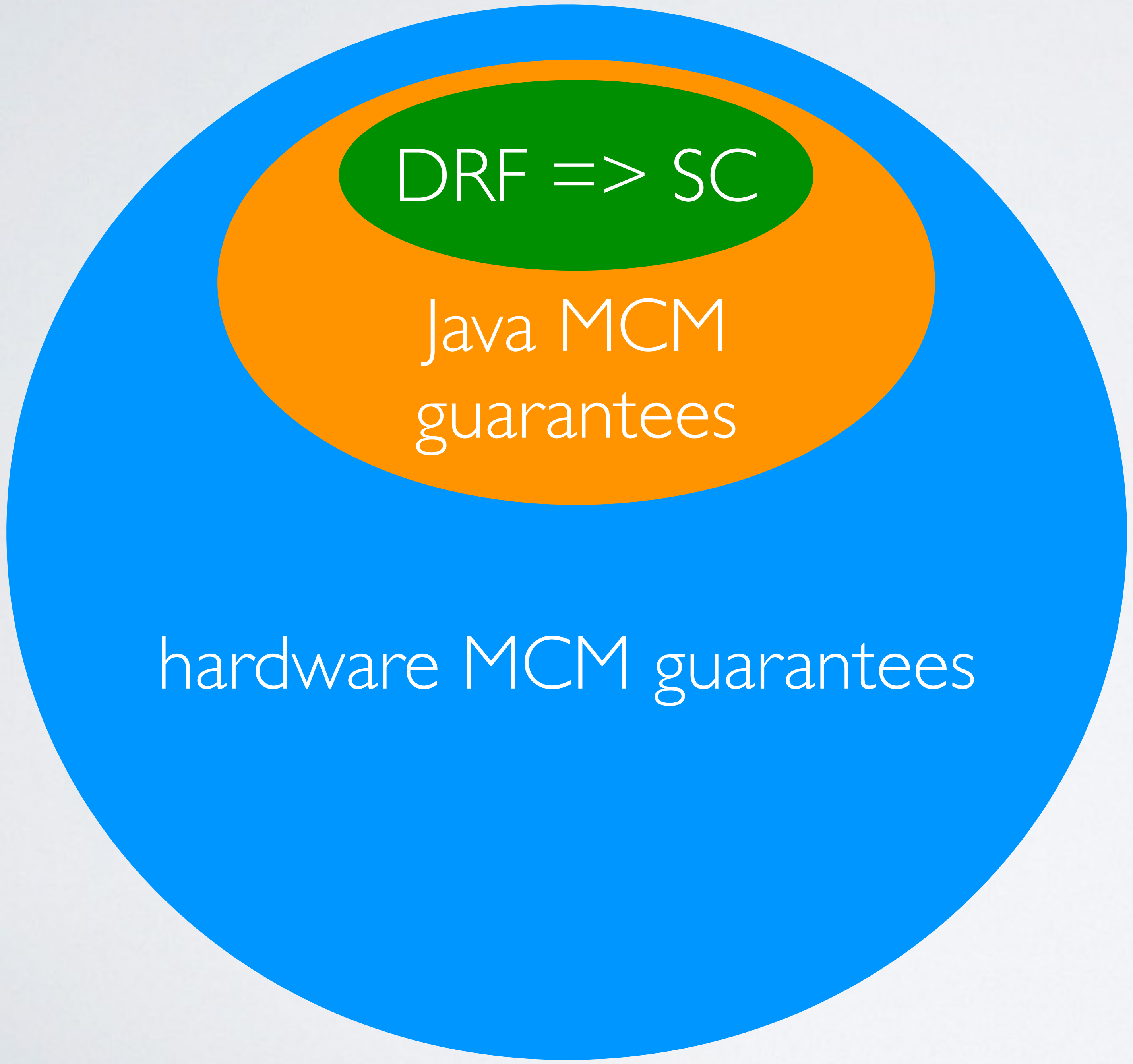
C, C++, JAVA

- compiler optimizations need reordering ability
    - e.g., register allocation, loop-invariant code motion
- these high-level languages have *very* relaxed MCMs
    - want to leave the door open for future compiler optimizations

SYNC IN C, C++, JAVA

- C, C++
  - **pthread_*** from <pthread.h>
  - **atomic_*** from <stdatomic.h> or <atomic>
    - more on this next week :-)
- Java
  - **volatile** field qualifier
  - **synchronized** blocks
  - fancier stuff in **java.util.concurrent** package

C, C++, JAVA => HW



everyone guarantees DRF => SC

if there is a race, guarantees vary

with a race, C/C++ semantics are **undefined**

with a race, Java preserves type safety

QUESTIONS

- Section 5.3 states that the XC implementation's reorder unit must ensure that "loads immediately see updates due to their own stores." Doesn't a load by definition not write to memory?

COMPILER VS HW FENCES

- How does a complier enforce FENCE instructions? How does a FENCE instruction know when each operation is done or ordered?

- In Power Memory Model, accesses can be made without FENCEs (if load L1 obtains a value used to calculate an effective address or data value of a subsequent store S2, then Power orders load L1 before store S2). In this case, when is the data dependency detected and instructions ordered? Is it speculative?

SYNCHRONIZATION CONFLICTS

- On page 66 & 67: Are conflicting synchronization operations what we want for synchronization? Could you explain the rules in more detail?

- What are transitive conflicts in synchronization operations? It was defined on page 67. Can you please give an example?

DRF => SC DEFINITIONS (1/2)

- Some memory operations are tagged as *synchronization* ("synchronization operations"), while the rest are tagged *data* by default ("data operations"). Synchronization operations include lock acquires and releases.
- Two data operations Di and Dj *conflict* if they are from different cores (threads) (i.e., not ordered by program order), access the same memory location, and at least one is a store.
- Two synchronization operations Si and Sj *conflict* if they are from different cores (threads), access the same memory location (e.g., the same lock), and the two synchronization operations are not compatible (e.g., acquire and release of a spinlock are not compatible, whereas two read_locks on a reader–writer lock are compatible).

DRF => SC DEFINITIONS (2/2)

- Two synchronization operations Si and Sj *transitively conflict* if either Si and Sj conflict or if Si conflicts with some synchronization operation Sk, Sk <p Sk' (i.e., Sk is earlier than Sk' in a core K's program order), and Sk' transitively conflicts with Sj.
- Two data operations Di and Dj *race* if they conflict and they appear in the global memory order without an intervening pair of transitively conflicting synchronization operations by the same cores (threads) i and j. In other words, a pair of conflicting data operations Di <m Dj are *not* a data race if and only if there exists a pair of transitively conflicting synchronization operations Si and Sj such that Di <m Si <m Sj <m Dj.
- An SC execution is data-race-free (DRF) if no data operations 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. This support usually requires some special actions for synchronization operations.

DR => WTF

- Why is it that once a data race occurs, SC is no longer valid in DRF and how do you reason about the undecidable halting problem even if its unsolvable
- Why is it the case that after data races, execution may no longer obey SC? Isn't the XC model still providing the illusion of SC, within the constraints of the program as it was written (i.e., there may be a fence missing that would enforce the behavior that the programmer expected, but from XC's point of view it is still following the rules)?

MCM TRANSLATION

- What would a programmer have to do to implement a stronger consistency model in code on a processor with weaker consistency?

- The paper mentions that with sufficient FENCES, a relaxed model like XC can appear to look like SC. At what point does FENCEs become prohibitively expensive and how can you qualitatively asses the impact of a FENCE?

HW MCM SPECS

- Why hasn't Intel or AMD not adopted a consistency model similar to "SC for DRF" yet? If implemented properly, it has the benefits of both SC as well as XC.
- Is the difficulty in comparing Power w.r.t. Alpha, ARM, RMO, and XC due to the difficulty in formalizing their specifications and constructing them into proofs?

PERFORMANCE

- Considering `HWSYNC` (IBM Power) basically synchronizes all cores, what is the performance impact compared to `FENCE` in other memory consistency models? How much lighter are `LWSYNC` instructions compared to `FENCE`?

- How does RC performance compare against TSO performance for languages with a SC memory model such as Java volatiles? When running mostly Java code, do the required fences with RC (lets say on ARM) diminish substantially the power and cost savings versus x86?

MCMs IN PRACTICE

- Has anyone ever studies the decrease in programmer productivity resulting from the non-intuitiveness of RC?
- At the end of 5.3, the authors discuss how they argued for a return to TSO or SC, but state that that did not happen. I thought previously we stated that Intel architecture uses a TSO-like memory model and relaxed models like Alpha died out. What are the authors referencing here when they say that architectures moved away from simpler consistency models?

QUESTIONS

- Some limits on relaxed memory ordering are present to prevent "astonishing" programmers, who typically expect sequential intra-thread execution. Are there programming models that don't presume sequentially executing threads that might be more amendable to very relaxed memory models?

- Do there exist any inter-core hardware synchronization instructions in modern architectures such as x86? e.g. shared hardware locks or semaphores?

- Would any of the previously discussed memory consistency models have benefits in GPU's?