RELAXED CONSISTENCY
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.

<table>
<thead>
<tr>
<th>Operation 1</th>
<th>Load</th>
<th>Store</th>
<th>RMW</th>
<th>FENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>Store</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>RMW</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>FENCE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
With the formalisms of the last section, we can now reveal why Section 5.2.2’s two examples work correctly. Figure 5.1 (a) shows an XC execution of the example from Table 5.3 in which core C1’s stores S1 and S2 are reordered, as are core C2’s loads L2 and L3. Neither reordering, however, affects the results of the program. Thus, as far as the programmer can tell, this XC execution is equivalent to the SC execution depicted in Figure 5.1 (b), in which the two pairs of operations are not reordered.

(a) An XC Execution
**TABLE 5.6:** Synchronization in TSO vs Synchronization in XC.

<table>
<thead>
<tr>
<th>Code</th>
<th>TSO</th>
<th>XC</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire lock</td>
<td>RMW: test-and-set L /* read L, write L=1 if L==1, goto RMW */ if lock held, try again</td>
<td>RMW: test-and-set L /* read L, write L=1 if L==1, goto RMW */ if lock held, try again <strong>FENCE</strong></td>
</tr>
<tr>
<td>critical section</td>
<td>loads and stores</td>
<td>loads and stores</td>
</tr>
<tr>
<td>release lock</td>
<td>store L=0</td>
<td><strong>FENCE</strong> store L=0</td>
</tr>
</tbody>
</table>
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

5.5.2 Causality and Write Atomicity

Here we illustrate two subtle properties of relaxed models. The first property, causality, requires that, “If I see it and tell you about it, then you will see it too.” For example, consider Table 5.9 where core C1 does a store \( S_1 \) to update data1. Let core C2 spin until it sees the results of \( S_1 \) \( (r_1 == \text{NEW}) \), perform FENCE \( F_1 \), and then do \( S_2 \) to update data2. Similarly, core C3 spins on load \( L_2 \) until it sees the result of \( S_2 \) \( (r_2 == \text{NEW}) \), performs FENCE \( F_2 \), and then does \( L_3 \) to observe store \( S_1 \). If core C3 is guaranteed to observe \( S_1 \) done \( (r_3 == \text{NEW}) \), then causality holds. On the other hand, if \( r_3 \) is 0, causality is violated.

The second property, write atomicity (also called store atomicity), requires that a core’s store is logically seen by all other cores at once. XC is write atomic by definition since its memory order \( (<m) \) specifies a logically atomic point at which a store takes effect at memory. Before this point, no other cores may see the newly stored value. After this point, all other cores must see the new value or the value from a later store, but not a value that was clobbered by the store. 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.

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

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Core C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1: \text{data1} = \text{NEW}; )</td>
<td>( L_1: r_1 = \text{data1}; ) \n ( B_1: \text{if (r1 \neq \text{NEW}) goto L1; ) \n ( \text{F1: FENCE} ) \n ( S_2: \text{data2} = \text{NEW}; )</td>
<td>( /\ast \text{Initially, data1 &amp; data2 = 0} /\ast )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( L_2: r_2 = \text{data2}; ) \n ( B_2: \text{if (r2 \neq \text{NEW}) goto L2; ) \n ( \text{F2: FENCE} ) \n ( L_3: r_3 = \text{data1}; /\ast r3==\text{NEW}? /\ast )</td>
</tr>
</tbody>
</table>
## INDEPENDENT READS, INDEPENDENT WRITES

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

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Core C3</th>
<th>Core C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: data1 = NEW;</td>
<td>S2: data2 = NEW;</td>
<td>L1: r1 = data1; /* NEW <em>/ &lt;br&gt; F1: FENCE &lt;br&gt; L2: r2 = data2; /</em> NEW? */</td>
<td>/* Initially, data1 &amp; data2 = 0 <em>/ &lt;br&gt; L3: r3 = data2; /</em> NEW <em>/ &lt;br&gt; F2: FENCE &lt;br&gt; L4: r4 = data1; /</em> NEW? */</td>
</tr>
</tbody>
</table>

*Note: The execution in Table 5.10 illustrates the PERSISTENCE of independent reads and independent writes.*
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 compiler 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?
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).
Two synchronization operations $S_i$ and $S_j$ *transitively conflict* if either $S_i$ and $S_j$ conflict or if $S_i$ conflicts with some synchronization operation $S_k$, $S_k < p S_k'$ (i.e., $S_k$ is earlier than $S_k'$ in a core $K$’s program order), and $S_k'$ transitively conflicts with $S_j$.

Two data operations $D_i$ and $D_j$ *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 $D_i < m D_j$ 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 $D_i < m S_i < m S_j < m D_j$.

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 it's 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 assess 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 studied 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?