Consistency Models

- UNDERSTANDING CONSISTENCY MODELS
  - Atomicity
  - Program Ordering
  - Visibility

- POPULAR CONSISTENCY MODELS
  - Sequential Consistency
  - IBM/370 Processor Consistency
  - SPARC TSO/PSO/RMO
  - Weak Ordering
  - PowerPC Weak Consistency

- VISIBILITY
- MEMORY REFERENCE REORDERING

Readings

- Readings:
  - Adve/Gharachorloo tutorial
  - Hill: position paper on simple models

Basics of Memory Consistency (Ordering)

Reorder load before store

Proc0

- at A=1
- if (load B==0) {
- ...critical section
- }

Proc1

- at B=1
- if (load A==0) {
- ...critical section
- }

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
- ISA must specify consistency model
- Common example using Dekker’s algorithm for synchronization
  - If load reordered ahead of store (as we assume for a baseline OOO CPU)
  - Both Proc0 and Proc1 enter critical section, since both observe that other’s lock variable (A/B) is not set
- If consistency model allows loads to execute ahead of stores, Dekker’s algorithm no longer works
  - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha

Sequential Consistency [Lamport 1979]

- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained
- Now Dekker’s algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs

High-Performance Sequential Consistency

- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)
High-Performance Sequential Consistency

- Load queue records all speculative loads
- Bus writes/updates are checked against LIQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive flush is OK

Relaxed Consistency Models

- Key insight: only synchronization references need to be ordered
- Hence, relax memory for all references
  - Enable high-performance OOO implementation
- Require programmer to label synchronization references
  - Hardware must carefully order these labeled references
  - All other references can be performed out of order
- Labeling schemes:
  - Explicit synchronization ops (acquire/release)
  - Memory fence or memory barrier ops:
    - All preceding ops must finish before following ones begin
- Often: fence ops cause pipeline drain in modern OOO machine

Why Relaxed Consistency Models?

- Original motivation
  - Allow in-order processors to overlap store latency with other work
    - “Other work” depends on loads, hence must let loads bypass stores and execute early:
    - Implement a store queue
  - This breaks sequential consistency assumption that all references are performed in program order
- This led to definition of processor consistency, SPARC TSO, IBM/370
  - All of these relax read-to-write program order requirement
- Subsequent developments
  - It would be nice to overlap latency of one store with latency of other stores
  - Allow stores to be performed out of order with respect to each other
  - This breaks SC even further
- This led to definition of SPARC PSO/RMO, WD, PowerPC WC, Itanium
- What’s the problem with relaxed consistency?
  - Shared memory programs can break if not written for specific cons. model

Understanding Consistency Models

- RAW dependencies to/from Proc0/Proc1 may or may not occur
  - Also, WAR and WAW dependencies may or may not occur
  - Relatively simple set of rules governs which (RAW/WAR/WAW) edges are required (must be observed), which ones are not required
- Observing certain edges provides visibility to other processors
  - Hence requires us to observe (some) subsequent edges
- Causality:
  - If I observe A, and B is ordered before A, I must also observe B
  - Without causality, system becomes virtually impossible to program

Constraint graph

- Defined for sequential consistency by Landin et al., ISCA-18
- Directed graph represents a multithreaded execution
  - Nodes represent dynamic instruction instances
  - Edges represent their transitive orders (program order, RAW, WAW, WAR)
- If the constraint graph is acyclic, then the execution is correct
  - Cycle implies A must occur before B and B must occur before A => contradiction

ECE 752: Advanced Computer Architecture I
Constraint graph example - SC

Proc 1

ST A

ST B

Proc 2

LD B

LD A

Cycle indicates that execution is incorrect

Anatomy of a cycle

Proc 1

Proc 2

ST A

ST B

LD A

LD B

Program order

Program order

Incoming invalidate

Program order

Cache miss

High-Performance Sequential Consistency

Load queue

System address bus

Bus writes

Bus upgrades

Other processors

Load queue records all speculative loads
Any matching load gets marked for replay
At commit, loads are checked and replayed if necessary
Practically, conflicts are rare, so expensive flush is OK

Understanding Relaxed Consistency

• Three important concepts
  – Atomicity
    • do writes appear at the same time to all other processors?
  – Program order
    • do my references have to be ordered with respect to each other?
  – Visibility (causality)
    • Does anyone care? This is the most subtle...

Possible Relancements

Relax Write to Read program order
Relax Write to Write program order
Relax Read to Read and Write to Write program order
Read others’ write early
Read own write early

From widely cited tech report:

Sequential Consistency

• Informally, all processors’ references are interleaved in total global order
• Multiple total global orders are possible and correct
  • Order determined in somewhat arbitrary manner
    – On bus-based SMP, by order in which bus is acquired
    – On directory-based system (like Origin 2000), order in which requests are asked (not order in which they arrive)
• All processors must maintain total order among their own references
• Again, key is to maintain illusion of order (visibility)
SC and ILP

- SC appears to preclude high performance, ILP (loads after stores)
- Can use speculation and prefetching to avoid these bottlenecks
  - Prefetch load data, store ownership early
  - as soon as address known and load/store issues in OOO core
- Hold off committing result until load/store has been ordered at commit
  - If conflicting remote event occurs before then, squash speculation
- If load, refetch instruction stream
- If store, fetch line ownership again

How to support speculation and rollback?
- Simple speculation within OOO window: MIPS R10000
- Aggressive speculation:
  - Speculative retirement [Ranganathan et al., ISPA 1997]
  - Speculative stores [Smith et al., ISCA 1999]
- Kilo-instruction checkpointing:
- Latencies growing to 100s of cycles, need potentially huge speculation buffers

Recent Trends

- Many are arguing that SC is best approach
  - ILP/speculation can be used to match performance of relaxed models
  - Adve, Falsafi, Hill all seem to be arguing this
- Is it really true? Conventional wisdom was that SC ordering rules must be relaxed to achieve performance
- Latencies relative to processor core are increasing
- Can massive speculation buffers really be justified/implemented?
- Reality:
  - All modern ISAs (Alpha, PowerPC, IA-64) have weak consistency models
  - Existence proof that programmers are willing to tackle the complexity
- Even less modern ISAs (IA-32, IBM/370) have relaxed models

Sequential Consistency

- No reordering allowed
- Writes must be atomic
  - Except can read own write early

IBM/370 Consistency

- Similar to IA-32
- Read->Write order relaxed
- Writes must occur atomically (others cannot read write early)
- Cannot read own write early! (not true for IA-32)

Processor Consistency

- Same as IBM/370, except writes not atomic
- Relax read to write order
- Writes need not occur atomically
SPARC TSO
- One of 3 SPARC consistency models determined by MSW mode bits
- This is the one actually used by real programs
- Reads may bypass writes
- Writes must be atomic

What Breaks?
- When relaxing read→write program order, what breaks?
  - Dekker's algorithm for mutual exclusion: initially A=B=0
  - Since read of B bypasses store of A on Proc0 (and vice versa on Proc1), mutual exclusion is no longer guaranteed:
    - Neither processor sees other's write since reads are moved up
    - Both processors believe they have mutually exclusive access
  - Fix?
    - Programmer must insert memory barriers between store and load
    - Force store to complete before load is performed
    - If stores not atomic (in PC), memory barrier must force atomicity

SPARC PSO
- SPARC second attempt at consistency model (not used)
- Reads may pass writes; writes may pass writes
- Writes must be atomic (cannot read other's write early)

What Breaks?
- When writes can pass writes in program order, what breaks?
  - Producer-consumer pattern (e.g. OS control block update)
    - Update control block, then set flag to tell others you are done with your update
    - Proc1 sees store of flag before it sees store of A, and reads stale copy of A
  - Fix?
    - Programmer must insert store memory barrier between two stores on Proc0
    - Hardware forces st A to complete (atomically) before st flag is performed

SPARC RMO
- SPARC third attempt at a relaxed consistency model
- Fully relaxed ordering
- Writes must still be atomic

Weak Ordering
- Equivalent to SPARC RMO
- Used by Alpha
- Fully relaxed ordering; writes must be atomic
Similar example as when writes can pass writes
• When reads can pass reads, what breaks?
  – Similar example as when writes can pass writes
  – Proc1 moves up read of A and reads stale value
  – Usually this requires branch prediction
• Branch misprediction recovery won’t help!
  – Branch was predicted correctly; flag was set to 0 by the time Proc1
  reads flag
• Fix?
  – Programmer must insert membar between two loads on Proc1 as well

When reads can pass reads, what else can break?
• Data dependence ordering is assumed even in weaker models
  – Typical use: create new linked list entry, initialize it, insert it on head of
    list
  – Force update of head of list to occur last (membar)
  – Expect that Proc1 won’t be able to dereference head until after its
    been updated due to data dependence between Id R1 and Id R2
• Wrong! What happens with value prediction?
  – Proc1 predicts value of R1, performs Id R2 with predicted R1, gets stale
data
  – Then, it validates R1 predicted value by performing load of head and
values match: no value misprediction! Yet Proc1 read stale value of A

What Else Breaks?
Proc2
st A=1
membar
st flag=0
Proc1
if (flag=0)
print A
Read of A
by Proc1
bypasses
read of flag

• When reads can pass reads, what breaks?
• When reads can pass reads, what else can break?
  – Data dependence ordering is assumed even in weaker models
  – Typical use: create new linked list entry, initialize it, insert it on head of
    list
  – Force update of head of list to occur last (membar)
  – Expect that Proc1 won’t be able to dereference head until after its
    been updated due to data dependence between Id R1 and Id R2
• Wrong! What happens with value prediction?
  – Proc1 predicts value of R1, performs Id R2 with predicted R1, gets stale
data
  – Then, it validates R1 predicted value by performing load of head and
values match: no value misprediction! Yet Proc1 read stale value of A

What Else Breaks?
Proc2
MTA=A
membar
st head=A
Proc1
Id R1=Head
Id R2=\therefore R1
Data dependence
prevents Id R2 from
bypassing Id R1

• When reads can pass reads, what else can break?
• When reads can pass reads, what else can break?
  – Data dependence ordering is assumed even in weaker models
  – Typical use: create new linked list entry, initialize it, insert it on head of
    list
  – Force update of head of list to occur last (membar)
  – Expect that Proc1 won’t be able to dereference head until after its
    been updated due to data dependence between Id R1 and Id R2
• Wrong! What happens with value prediction?
  – Proc1 predicts value of R1, performs Id R2 with predicted R1, gets stale
data
  – Then, it validates R1 predicted value by performing load of head and
values match: no value misprediction! Yet Proc1 read stale value of A

What Breaks?
Proc0
st A=1
Proc1
while (B==0);
print A

• When stores are no longer atomic, what breaks?
  – 3-processor example required to demonstrate transitivity:
    – Proc0 writes B before it sees Proc0’s write of A
    – Proc0 reads A after it sees Proc0’s write of B
    – Proc1 gets stale copy of A since write from Proc0 hasn’t arrived yet
• Fix?
  – Proc2’s read of A must be an atomic RMW operation (or II/sc), which
    will force it to be ordered after Proc0’s write of A
  – Note that a membar at Proc1 or a membar at Proc0 do not help

PowerPC Weak Ordering

• Fully relaxed ordering
• Writes need not be atomic

How Do We Synchronize?
• With SC, synchronization can be accomplished with e.g. Dekker’s
  algorithm, which relies on store->load ordering
• With weaker models, synchronization operations may need to be
  explicitly identified to the processor.
• Processor then treats synchronization operations with stricter rules
  – E.g. release consistency (RC) uses explicit “acquire” and “release”
    primitives which are strongly ordered, while standard loads and stores
    are weakly ordered
  – Acquire and release protect mutually exclusive regions (critical sections)
    These impose ordering fences or barriers on other memory operations, which
    are otherwise unordered.
  – Acquire: full memory barrier, all previous loads and stores ordered with
    respect to all subsequent loads and stores, all remote stores must be
    visible to subsequent loads
  – Release: write memory barrier, all previous stores ordered with respect
    to all subsequent stores (i.e. all critical section updates visible to
    everyone before release visible).
Release Consistency

- Acquire/Release pairs protect critical sections
- Without special semantics for acquire/release
  - load A may not see st A due to relaxed ordering
- Instead:
  - Proc0 acquire forces all prior writes to be visible to all other processors
  - Proc1 acquire prevents subsequent reads or writes from being performed before acquire has completed
- In proposed RC systems, acquire and release are special instructions:
  - Hardware knows to treat them with stricter ordering rules
  - Special acquire/release instructions are not strictly necessary

Synchronization in Weak Models

- RC not actually implemented in any cache-coherent hardware
  - Lots of proposals for RC variants in software-based DSM (SVM)
- RC can be approximated in weakly consistent systems by providing two flavors of memory barrier instructions
  - Acquire: corresponds to full memory barrier (Alpha membar, PowerPC sync)
  - Release: corresponds to store memory barrier (Alpha membar, PPC lwsync)
- Memory barriers after lock acquire and before lock release achieve benefits of release consistency

Synchronization

- Burden is on programmer to protect all shared accesses with locks, critical sections, and use acquire/release primitives
  - If no acquire/release or membar instructions, then what?
  - Usually fall back on atomic RMW instructions (compare-and-swap)
    - These either have special ordering semantics, or force ordering because they do both a read and a write simultaneously
    - In 370/TSO/A32, many sync. primitives (e.g. A/c spin loops) work without barriers (barriers are implicit)
- Bottom line: can’t write correct shared-memory programs in WC systems without synchronization!
  - WC rules allow arbitrary delays, meaning other processors may never see your writes
  - Synchronization ops and memory barriers force writes to be visible to other processors

What About Visibility/Causality?

- None of these definitions clarify what is meant by visibility (i.e. ISA spec says something like “it must appear as if R->W order is maintained…”)
- What does this mean?
  - The programmer must not be able to detect that references have been reordered
  - Or, bounds must be set on how much “slack” each reference has
  - Also known as causality
- Construct a constraint graph:
  - Identify all (PO,RAM,WAR,IAW) edges that have occurred between processors (i.e. all dependences that have been observed in the past)
  - These are what indicate visibility or causality to another processor’s references
  - Then determine which (if any) prior (RAM,WAR,IAW) edge implies causality (else cycle will form)
- The rules for which types of PO (program order) edges are present depend on the consistency model’s relaxation of rd->rd, wr->wr, rd->wr, etc.

4 Simple Steps to Understanding Causality

1. Single mem ref/CPU
  - Load can use any version (bind to …)
  - Stores? Coherence requires total order per address: A₀, A₁, A₂

<table>
<thead>
<tr>
<th>PO</th>
<th>PL</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₂</td>
</tr>
<tr>
<td>A₁</td>
<td>A₀</td>
<td>A₂</td>
<td>A₁</td>
</tr>
<tr>
<td>A₀</td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
</tr>
</tbody>
</table>

© 2005 Mikko Lipasti
4 Simple Steps to Understanding Causality

2. Two or more mem refs to same address/CPU:

<table>
<thead>
<tr>
<th>Case</th>
<th>OK</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
<tr>
<td>ii.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
<tr>
<td>iii.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
<tr>
<td>iv.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
</tbody>
</table>

SC ordering rules imply that all prior stores from Proc0 must now be visible from Proc1. Hence, Proc0 must now be visible to Proc1.

4. Causality extends transitively across all memory locations and all processors in the system

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>st A1</td>
<td>st B1</td>
<td>Id B1</td>
<td>Causal RAW</td>
</tr>
<tr>
<td>st C1</td>
<td>Id C2</td>
<td>Implies A1 or newer</td>
<td></td>
</tr>
<tr>
<td>Id B2</td>
<td>Id B2</td>
<td>Implies B1 or newer</td>
<td></td>
</tr>
</tbody>
</table>

4 Simple Steps to Understanding Causality

3. Two or more mem refs to diff address/CPU:

<table>
<thead>
<tr>
<th>Case</th>
<th>OK</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Id A</td>
<td>Id B</td>
</tr>
<tr>
<td>ii.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
<tr>
<td>iii.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
<tr>
<td>iv.</td>
<td>Id A</td>
<td>Id A</td>
</tr>
</tbody>
</table>

Causality through consistency kicks in:
two addresses are now synchronized

Causality Example

- Assuming SC
  - stB > ldA RAW edge was observed (i.e. Proc0/ldB got its value from Proc0)
  - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1, Proc0/stA is upper bound on slack for ldA
  - Hence, Proc1/ldA must get its value from Proc0/stA

- In MIPS R10000, ldA can be reordered (issued) ahead of ldB
  - However, ldA is retired after ldB. Since stA/ldB are retired in order at Proc0, we know that Proc1/stB had to occur after ldB (otherwise ldB would get refetched). Hence stA will have reached Proc1 before ldA can retire
  - Hence, violation is detected and ldA reissues

- For this to work, the write to A must complete before ldB retires!

Causality Example 2

- More subtle case (Jeuexké’s algorithm), again assuming SC
  - ldB > stB WAR edge was observed (i.e. Proc0/ldB got its value from Proc0)
  - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1, Proc0/stA is upper bound on slack for Proc1/ldA
  - Hence, Proc1/ldA must get its value from Proc0/stA

- In MIPS R10000, ldA can be reordered (issued) ahead of stB
  - However, ldA is retired after stB. Since stA/ldB are retired in order at Proc0, we know that Proc1/stB had to occur after ldB (otherwise ldB would get refetched). Hence stA will have reached Proc1 before ldA can retire
  - Hence, violation is detected and ldA reissues

- For this to work, the writes to A and B must be ordered!

Causality Example 3

- More subtle case, again assuming SC
  - stB > stB WAW edge was observed (i.e. Proc1/stB was ordered after Proc0/stB)
  - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1, Proc0/stA is upper bound on slack for Proc1/stA
  - Hence, Proc1/stA must get its value from Proc0/stA

- In MIPS R10000, ldA can be reordered (issued) ahead of stB
  - However, Proc0/stA is retired before Proc1/stB. Since Proc1/stB occurs after Proc0/stB, and ldA retires after Proc1/stB, we know that Proc0/stA had to reach Proc1 before A can retire
  - Hence, violation is detected and ldA reissues

- For this to work, the writes of A and B must be ordered!
What About Weaker Models?

![Diagram](image)

- Causality rules are in fact very similar, only the granularity changes.
- In WC, upper bound on slack for a load is not the store preceding an observed RAW/WAR/WAW edge, but the store (if any) preceding the membar before the observed RAW/WAR/WAW edge preceding your latest membar.
  - If either membar absent, ProCol/st A is not the upper bound on slack for Id A
  - Would have to search backward in ProCol to find st A prior to latest membar preceding st B
  - Any edges in same “epoch” (after latest membar) don’t matter until next “epoch”

Memory Reference Reordering

- Can happen inside the core
  - OOO issue of loads
- Can happen outside the core
  - Store queue, write thru queue can reorder stores
  - Interconnect, routing, message/snoop queues
    - Messages arrive or are processed out of order
- Correct implementation must consider both
  - Coherence: total order to same address
  - Consistency: order across addresses, atomicity
- **What we must know: when is each store complete**
  - No more stale copies of block exist in any cache

Coherence Ordering

- Stores to same address must have total order
  - Shared bus is easy: arb order or resp order
  - Ring is fairly easy (later)
  - Interconnection networks are harder (later)
- Loads to same address must follow program order: load-to-load order
  - Track younger speculative loads
  - Replay if remote store could change loaded value

Consistency Ordering

- Store-store order (if required)
  - Retire stores in order
  - Prefetch exclusive permission OOO for performance
  - In weak models, order across members only
- Store-load order (if required)
  - Retire stores in order
  - Read-set tracking for speculative loads
  - In weak models, inhibit speculation across members (or use membar-aware read-set tracking)
- Write atomicity (if required)
  - Don’t source dirty block until store is complete

Reordering Inside Core

- Rely on in-order commit of loads and stores
- **Read-set tracking** for load-load coherence and store-load ordering
  - Track speculative loads using load queue
  - Check “older” remote writes against load queue
  - Or check for load-hit-younger in insulated load queue
  - Replay speculative loads on violation to force new value
- Or, value-based consistency [Cain 2004]
  - **Replay loads in order @ commit, compare values**
  - Seems expensive, but simple filters avoid 97% of checks
  - No reorder, no recent miss, no recent snoop
Reordering Outside the Core

• Easy case: single shared bus
  – Arb or resp order determines write completion
  – This order immediately visible to all cores
• Multiple address-interleaved buses
  – Coherence (same address) still easy (same bus)
  – For consistency (diff addresses) can use implied order across buses (Q0 before Q1 before Q2 ...)
  – Otherwise have to collect ACKs (later)

Ring Order

• Req and resp traverse ring in order
  – Either process snoop, then forward req/resp, or
  – Eager forward with trailing resp (2x traffic)
• Races can be resolved given ring order
  – Not as simple as bus order
  – Can use home node as ordering point; extra latency since req is not active till after resp circulates
  – Can make reqs immediately active: retries
  – Or can reorder based on ring order [Marty ’06]
    • Simpler form of write-string forwarding

Network Reordering

• Deterministic routing provides pt-to-pt order
  – Always follow same path from A to B: FIFO
  – Messages leave A and arrive at B in same order
  – Ordering point can shift to A (e.g. directory)
• Indirect network (e.g. tree)
  – May have a central ordering point (tree root)
  – Ordering can shift to that point
• General case: no guarantees
  – E.g. adaptive routing from A to B
  – Or independent address-interleaved queues (Power4)

Physical vs. Logical Time

• Physical time systems
  – Ordering implies placement in physical time
  – Easier to reason about, observe, check, trace, replay
  – Less concurrency exposed, worse performance
• Logical time systems
  – Ordering is only relative, not in physical time
  – Based on causal relationships
    • Make sure INV from A->B stays ahead of data from A->B
  – Rely on ordering properties of interconnect, e.g. FIFO pt-to-pt order
  – Much harder to reason about, observe, check, trace, replay
  – Expose more concurrency, provide better performance

Interconnect Not Ordered

• How to detect write completion?
• Must collect ACKs for write misses, upgrades
  – ACK from all sharers indicates INV applied
    • Broadcast or multicast
    • Use directory sharing list (if it exists)
  – Proves no stale copies of block in the system
  – Can safely retire store (or membar)
• Physical time

ACK Collection

• Eager ACK once invalidates are ordered
  – Pass through ordering point (root of tree), or
    • Alphaserver G5320
  – Entered in FIFO queue or FIFO network lane or bus or ...
  – Don’t need to be applied just ordered
  – Must prevent subsequent reordering (FIFO)
  – Logical time
• Coarse-grained ACKs
  – Once per membar [IBM Power4] : physical time
  – VCT Coherence [Enright Jerger 2008]
    • ACK per memory region
    • Don’t source dirty blocks from region until ACK done
    • Logical time
Ordering Recap

- Inside core: in-order or read-set tracking
- Outside core: detect write completion
- For coherence, enforce:
  - Write order to same address
  - Read order to same address
- Per consistency model, enforce:
  - Write-write order across addresses
  - Write-read order across addresses
  - Write atomicity
- What core must know: when write is complete

Summary – Consistency Models

- SC simpler for programmers to understand
- Relaxed consistency models originally proposed to enable high performance with in-order processors (overlap store latency)
  - Most modern ISAs specify WC: Alpha, PowerPC, IA-64
  - Many claim that much of the performance benefit of relaxed consistency can be obtained with aggressive speculation
    - It is unclear whether or not this is true
    - Power cost of speculation buffering may be too high
  - Not too difficult to meet sufficient conditions by a combination of
    - speculation
    - bus ordering or detection of write completion, and
    - support for rollback
- Physical vs. logical time systems
  - Physical much easier, less concurrent
  - Logical much more difficult, more concurrent