

ECE/CS 757: Advanced Computer Architecture II

Instructor: Mikko H Lipasti

Spring 2009
University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, and Jim Smith, Natalie Enright Jerger, and probably others

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**

© 2005 Mikko Lipasti

2

Readings

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

3

Basics of Memory Consistency (Ordering)

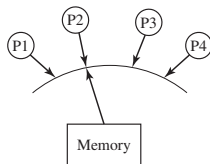
	Proc0	Proc1
Reorder		
load	st A=1	st B=1
before	if (load B==0) {	if (load A==0) {
store	...critical section	...critical section
	}	}

- 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

© 2005 Mikko Lipasti

4

Sequential Consistency [Lampport 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

© 2005 Mikko Lipasti

5

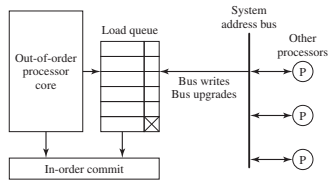
High-Performance Sequential Consistency

- 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)

© 2005 Mikko Lipasti

6

High-Performance Sequential Consistency



- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- 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

© 2005 Mikko Lipasti

7

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

© 2005 Mikko Lipasti

8

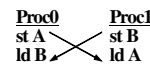
Why Relaxed Consistency Models?

- 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, WO, PowerPC WC, Itanium
- What’s the problem with relaxed consistency?
 - Shared memory programs can break if not written for specific cons. model

© 2005 Mikko Lipasti

9

Understanding Consistency Models

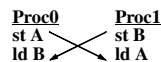


- Consistency model defines how memory references are ordered across processors/threads
 - Part of the instruction-set architecture
- PowerPC, SPARC, IA-32, etc. each have their own consistency model
 - In some cases, more than one!
- The program semantics will change with consistency model
 - More relaxed consistency models enable more “correct” outcomes
 - Even strict consistency models allow multiple correct outcomes

© 2005 Mikko Lipasti

10

Understanding Consistency Models



- RAW dependences to/from Proc0/Proc1 may or may not occur
 - Also, WAR and WAW dependences 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

© 2005 Mikko Lipasti

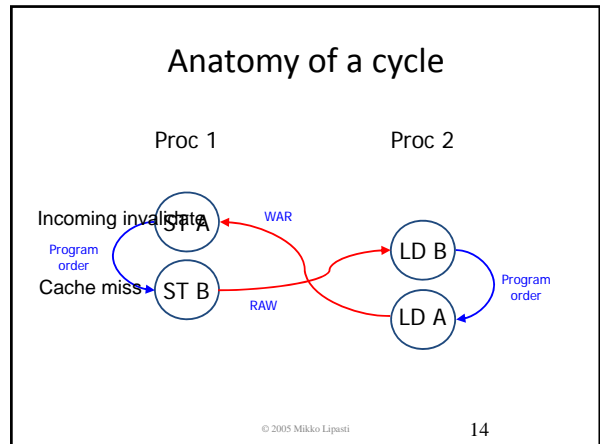
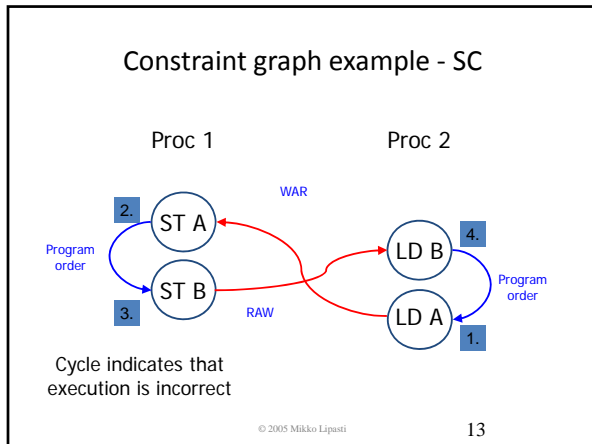
11

Constraint graph

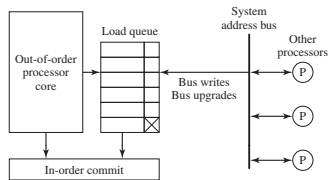
- Defined for sequential consistency by Landin et al., ISCA-18
- Directed graph represents a multithreaded execution
 - Nodes represent dynamic instruction instances
 - Edges represent their transitive orders (program order, RAW, WAW, WAR).
- **If the constraint graph is acyclic, then the execution is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

© 2005 Mikko Lipasti

12



High-Performance Sequential Consistency



- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- 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

© 2005 Mikko Lipasti

15

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?

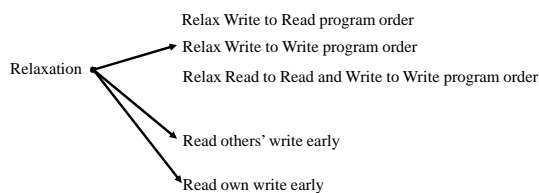
– Visibility (causality)

- Does anyone care? This is the most subtle...

© 2005 Mikko Lipasti

16

Possible Relaxations

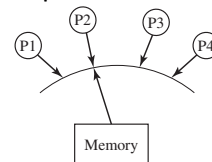


- From widely cited tech report:
 - [Adve/Gharachorloo, "Shared Memory Consistency Models: A Tutorial"]

© 2005 Mikko Lipasti

17

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 acked (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)

© 2005 Mikko Lipasti

18

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
 - conflicting event == invalidate message for matching address
- If load, refetch instruction stream
- If store, fetch line ownership again

© 2005 Mikko Lipasti

19

SC and ILP

- How to support speculation and rollback?
 - Simple speculation within OOO window: MIPS R10000
 - Aggressive speculation:
 - Speculative retirement [Ranganathan et al., SPAA 1997]
 - Speculative stores [Gniady et al., ISCA 1999]
 - Kilo-instruction checkpointing:
 - Safetynet [Sorin Ph.D. thesis, U. Wisconsin]
- Latencies growing to 100s of cycles, need potentially huge speculation buffers

© 2005 Mikko Lipasti

20

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

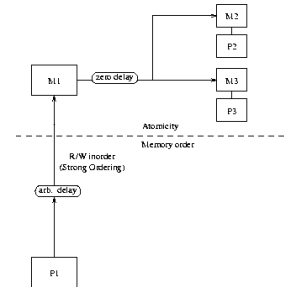
© 2005 Mikko Lipasti

21

Sequential Consistency

Sequential Consistency

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

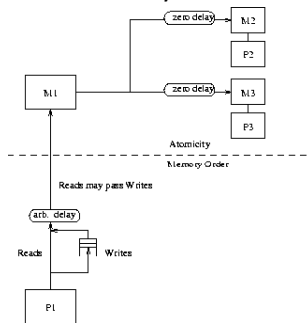


© 2005 Mikko Lipasti

22

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)

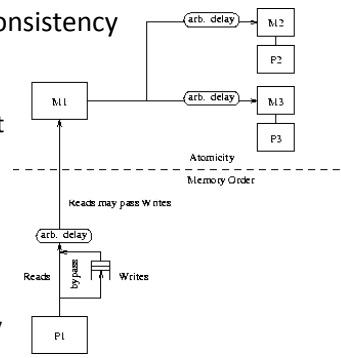


© 2005 Mikko Lipasti

23

Processor Consistency

- Same as IBM/370, except writes not atomic
- Relax read to write order
- Writes need not occur atomically



© 2005 Mikko Lipasti

24

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

© 2005 Mikko Lipasti 25

What Breaks?

```

Proc0
load  st A=1
before if (load B==0) {
store  ...critical section
}

Proc1
store st B=1
if (load A==0) {
...critical section
}

```

- 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

© 2005 Mikko Lipasti 26

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)

© 2005 Mikko Lipasti 27

What Breaks?

```

Store flag bypasses store of A
Proc0 st A=1
st flag=0

Proc1 while (flag==1) {};
print A

```

- 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

© 2005 Mikko Lipasti 28

SPARC RMO

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

© 2005 Mikko Lipasti 29

Weak Ordering

- Equivalent to SPARC RMO
- Used by Alpha
- Fully relaxed ordering; writes must be atomic

© 2005 Mikko Lipasti 30

What Breaks?

Proc0
st A=1
membar
st flag=0

Proc1
if (flag==0) ↑
print A Read of A
 bypasses
 read of flag

- 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

© 2005 Mikko Lipasti

31

What Else Breaks?

Proc0
st *A=1
membar
st head=A

Proc1
ld R1=head
ld R2=*R1 Data dependence
 prevents ld R2 from
 bypassing ld R1

- 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 ld R1 and ld R2
- Wrong! What happens with value prediction?
 - Proc1 predicts value of R1, performs ld 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

© 2005 Mikko Lipasti

32

What Else Breaks?

Proc0
st *A=1
membar
st head=A

Proc1
ld R1=head
ld R2=*R1 Data dependence
 prevents ld R2 from
 bypassing ld R1

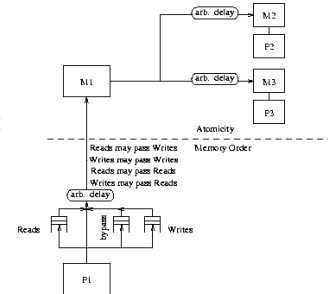
- Fix?
 - Programmer must insert membar between two loads on Proc1
- Cannot rely on data dependence ordering of loads
 - PowerPC OK (fixed in latest ISA spec, previously undefined)
 - Alpha requires membar (will break without it) -- expensive!
- Example of incomplete ISA definition due to assumptions about implementation
 - Assume no value prediction!
- Ref: [Martin et al., MICRO 2001]

© 2005 Mikko Lipasti

33

PowerPC Weak Ordering

- Fully relaxed ordering
- Writes need not be atomic



© 2005 Mikko Lipasti

34

What Breaks?

Proc0
st A=1

Proc1
while (A==0);
st B=1

Proc2
while (B==0);
print A
(st A arrives late at Proc2)

- When stores are no longer atomic, what breaks?
 - 3-processor example required to demonstrate transitivity:
 - Proc1 writes B after it sees Proc0's write of A
 - Proc2 reads A after it sees Proc1's write of B
 - Proc2 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 ll/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

© 2005 Mikko Lipasti

35

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).

© 2005 Mikko Lipasti

36

Release Consistency

Proc0
acquire (lock)
st A
release (lock)

Proc1
acquire(lock)
load A
store B
release (lock)

→

- 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 release forces all prior writes to be visible to all other processors before lock release is visible to anyone
 - Furthermore, 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

© 2005 Mikko Lipasti 37

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 stmembar, PPC lwsync)
- Memory barriers after lock acquire and before lock release achieve benefits of release consistency

© 2005 Mikko Lipasti 38

Synchronization in Weak Models

Proc0
ll/sc pin lock
membar
st A
stmembar
st lock
ld X

Proc1
ll/sc spin lock
membar
load A
st B
stmembar
st lock

↑

Later un-related load can still move up

- Acquire/release are programmer-annotated with appropriate membar

© 2005 Mikko Lipasti 39

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/IA-32, many sync. primitives (e.g. ll/sc 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

© 2005 Mikko Lipasti 40

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,RAW,WAR,WAW) edges that have occurred between processors (i.e. all dependencies 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 (RAW,WAR,WAW) 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.

© 2005 Mikko Lipasti 41

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: A_0, A_1, A_2

P0	P1	P2	P3
ld A_{0-2}		ld A_{0-2}	
	st A_1		
			st A_2

© 2005 Mikko Lipasti 42

4 Simple Steps to Understanding Causality

2. Two or more mem refs to same address/CPU:
Causality through coherence kicks in: two refs must interleave into global order correctly

- ld A – ld A
- ld A – st A
- st A – ld A
- st A – st A

Case	OK	!OK	Note
i	ld A ₁	ld A ₂	
	ld A ₃	ld A ₁	A ₂ implies A _{2,3}
ii	ld A ₀	ld A ₂	
	st A ₂	st A ₁	A ₂ implies A ₃
iii	st A ₂	st A ₂	
	ld A ₃	ld A ₁	Also 1T RAW
iv	st A ₁	st A ₂	
	st A ₃	st A ₁	Also 1T WAW

Store Order
A ₀
A ₁
A ₂
A ₃

43

4 Simple Steps to Understanding Causality

3. Two or more mem refs to diff address/CPU:
Causality through consistency kicks in: two addresses are now synchronized

- ld A – ld B
- ld A – st B
- st A – ld B
- st A – st B

Case	OK	!OK	Note
i	ld A ₁	ld A ₂	
	ld B ₃	ld B ₁	A ₂ implies B _{2,3}
ii	ld A ₀	ld A ₂	
	st B ₂	st B ₁	A ₂ implies B ₃
iii	st A ₂	st A ₂	
	ld B ₃	ld B ₁	A ₂ implies B ₂
iv	st A ₀	st A ₂	
	st B ₃	st B ₁	A ₂ implies B ₃

Store Order	
A ₀	B ₀
A ₁	B ₁
A ₂	B ₂
A ₃	B ₃

44

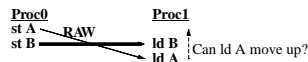
4 Simple Steps to Understanding Causality

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

P0	P1	P2	Notes
st A ₁			
st B ₁			
	ld B ₁		Causal RAW
	st C ₁		
		ld C ₁	Causal RAW
		ld A ₂	Implies A ₁ or newer
		ld B ₂	Implies B ₁ or newer

45

Causality Example

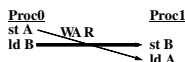


- Assuming SC
 - st B -> ld B RAW edge was observed (i.e. Proc1/ld B got its value from Proc0)
 - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1; Proc0/st A is upper bound on slack for ld A
 - Hence, Proc1/ld A must get its value from Proc0/st A
- How does the hardware track this?
 - In MIPS R10000, ld A can be reordered (issued) ahead of ld B
 - However, ld A is retired after ld B. Since st A/st B are performed in order at Proc0, the fact that Proc1 observed st B before retiring ld B implies it observed st A before it retires ld A.
 - Hence, violation is detected and ld A reissues
- For this to work, writes to A and B must be ordered!

© 2005 Mikko Lipasti

46

Causality Example 2

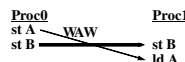


- More subtle case (Dekker's algorithm), again assuming SC
 - ld B -> st B WAR edge was observed (i.e. Proc0/ld B did not get its value from Proc1/st B)
 - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1; Proc0/st A is upper bound on slack for Proc1/ld A
 - Hence, Proc1/ld A must get its value from Proc0/st A
- How does the hardware track this?
 - In MIPS R10000, ld A can be reordered (issued) ahead of st B
 - However, ld A is retired after st B. Since st A/ld B are retired in order at Proc0, we know that Proc1/st B had to occur after ld B (o/wise ld B would get refetched). Hence st A will have reached Proc1 before ld A can retire
 - Hence, violation is detected and ld A reissues
- For this to work, the write to A must complete before ld B retires!

© 2005 Mikko Lipasti

47

Causality Example 3

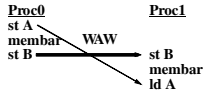


- More subtle case, again assuming SC
 - st B -> st B WAW edge was observed (i.e. Proc1/st B was ordered after Proc0/st B)
 - SC ordering rules imply that all prior stores from Proc0 must now be visible to Proc1; Proc0/st A is upper bound on slack for ld A
 - Hence, Proc1/ld A must get its value from Proc0/st A
- How does the hardware track this?
 - In MIPS R10000, ld A can be reordered (issued) ahead of st B
 - However, Proc0/st A is retired before Proc0/st B. Since Proc1/st B occurs after Proc0/st B, and ld A retires after Proc1/st B, we know that Proc0/st A had to reach Proc1 before ld A can retire
 - Hence, violation is detected and ld A reissues
- For this to work, the writes of A and B must be ordered!

© 2005 Mikko Lipasti

48

What About Weaker Models?

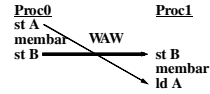


- Causality rules are in fact very similar, only the granularity changes
- In WC, upper bound on slack for a load is not the store preceding an observed RAW/WAR/WAW edge, but the store (if any) preceding the membar before the observed RAW/WAR/WAW edge preceding your latest membar.
 - If either membar absent, Proc0/st A is not the upper bound on slack for ld A
 - Would have to search backward in Proc0 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”

© 2005 Mikko Lipasti

49

What About Weaker Models?



- In 370/TSO (relaxed read-to-write), upper bound on slack for a load depends only on observed RAW edges, since WAR/WAW edges terminate in stores, and loads are allowed to bypass stores
 - Hence any tracking mechanism would only consider observed RAW edges
- Need Proc1 membar above to force ld A to see effects of st A

© 2005 Mikko Lipasti

50

Memory Reference Reordering

- Can happen inside the core
 - OOO issue of loads
- Can happen outside the core
 - Store queue, writethru 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

51

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

52

Consistency Ordering

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

53

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

54

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)

55

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*

56

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)

57

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

58

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

59

ACK Collection

- Eager ACK once invalidates are **ordered**
 - Pass through ordering point (root of tree), or
 - Alphaserver GS320
 - 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

60

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**

61

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

© 2005 Mikko Lijeski

62