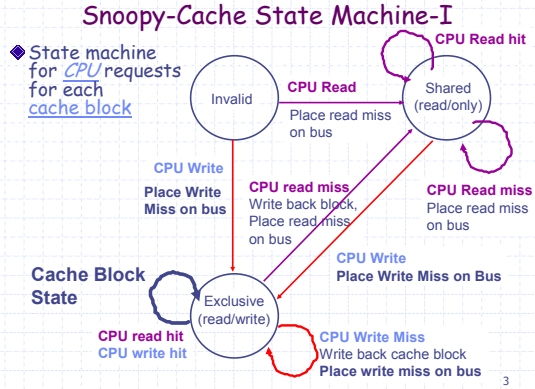
Cache Coherence and Memory Consistency

An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

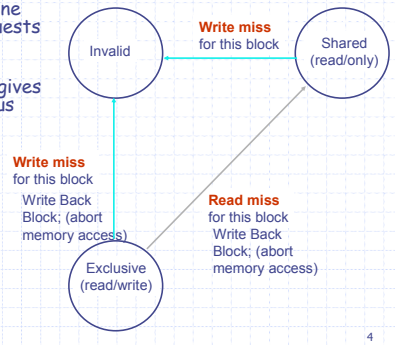
Snoopy-Cache State Machine-I

- State machine for *CPU* requests for each cache block



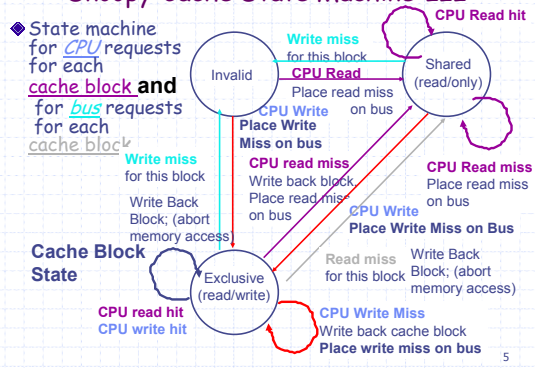
Snoopy-Cache State Machine-II

- State machine for *bus* requests for each cache block
- Appendix I gives details of bus requests



Snoopy-Cache State Machine-III

- State machine for *CPU* requests for each cache block **and** for *bus* requests for each cache block



Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

What happen if P1 reads A1 at this time?

Implementation Snoop Caches

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.

7

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache

8

MESI Protocol

- Simple protocol drawbacks: When writing a block, send invalidations even if the block is used privately
- Add 4th state (MESI)
  - **Modfied (private,!=Memory)**
  - **eXclusive (private,=Memory)**
  - **Shared (shared,=Memory)**
  - **Invalid**

Original Exclusive => Modified (dirty) or Exclusive (clean)

9

MESI Protocol

From local processor *P's* viewpoint, for each cache block

- **M**odified: Only *P* has a copy and the copy has been modifed; must respond to any read/write request
- **E**xclusive-clean: Only *P* has a copy and the copy is clear; no need to inform others about further changes
- **S**hared: Some other machines *may* have copy; have to inform others about *P*s changes
- **I**nvalid: The block has been invalidated (possibly on the request of someone else)

10

Memory Consistency

- Sequential Memory Access on Uniprocessor execution

```
A ← 10;        // First Write to A
A ← 20;        // Last write to A
Read A;        // A will have value of 100
```

If "Read A" returns value 100, the execution is wrong!

- Memory Consistency on Multiprocessor

P1	P2	P3	P4
Initial: A=B=0;			
A ← 10;	A==10	A==10	A==0
B ← 20;	B==20	B==0	B==20
	(Right)	(Right)	(Wrong?!)

What was expected?

11

Sequential Consistency

Sequential consistency: All memory accesses are in program order and globally serialized, or

- Local accesses on any processor is in program order
- All memory writes **appear** in the same order on all processors

Any other processor perceives a write to *A* only when it reads *A*

Programmer's view about consistency: how memory writes and reads are ordered on every processor

Programmer's view on P3	Programmer's view on P4
A←10;	B←20;
Read A (A==10);	Read A (A==0);
Read B (B==0);	Read B (B==10);
B←20;	A←10;
(Consistent)	(Inconsistent!)

12

Sequential Consistency

Consider writes on two processors:

P1:	A ← 0;	P2:	B ← 0;
	.....		.....
	A ← 1;		B ← 1;
L1:	if (B == 0) ...	L2:	if (A == 0) ...

Is there an explanation that L1 is true and L2 is false?

Global View	View from P1	View from P2
A ← 0	A ← 0	A ← 0
B ← 0	B ← 0	B ← 0
A ← 1	A ← 1	A ← 1
P1 Reads B	L1: Read B==0	---
P2 Reads A	---	L2: Read A==1
B ← 1	B ← 1	B ← 1

What is wrong if both statements (L1 and L2) be true?

- Can you find an explanation?
- If not, how would you prove there is no valid explanation?

Sequential Consistency Overhead

What could have been wrong if both L1 and L2 are true?

P1:	A ← 0;	P2:	B ← 0;
	.....		.....
	A ← 1;		B ← 1;
L1:	if (B == 0) ...	L2:	if (A == 0) ...

- A's invalidation has not arrived at P2, and B's invalidation has not arrived at P1
- Reading A or B happens before the writes

Solution I: Delay ANY following accesses (to the memory location or not) until an invalidation is ALL DONE.

Overhead:

- What is the full latency of invalidation?
- How frequent are invalidations?
- How about memory level parallelism?

Memory Consistence Models

Why should sequential consistency be the only correct one?

- It is just the most simple one
- It was defined by Lamport

Memory consistency models: A contract between a multiprocessor builder and system programmers on how the programmers would reason about memory access ordering

Relaxed consistency models: A memory consistency that is weaker than the sequential consistency

- Sequential consistency maintains some total ordering of reads and writes
- Processor consistency (total store ordering): maintain program order of writes from the same processor
- Partial store order: writes from the same processor might not be in program order

Memory Consistency Models

P1:	A ← 0;	P2:	B ← 0;
	.....		.....
	A ← 1;		B ← 1;
L1:	if (B == 0) ...	L2:	if (A == 0) ...

Explain in processor consistency that both L1 and L2 are true:

View from P1	View from P2	Another view from P2
A ← 0	B ← 0	A ← 0
B ← 0	B ← 1	B ← 0
A ← 1	A ← 0	L2: Read A==0
L1: Read B==0	L2: Read A==0	A ← 1
B ← 1	A ← 1	B ← 1
(a)	(b)	(c)

(b) Remote writes appear in a different order

(c) Local reads bypasses local writes (relax W->R order)

- Key point: programmers know how to reason about the shared memory

Memory Consistency and ILP

- Speculate on loads, flush on possible violations
  - With ILP and SC what will happen on this?

P1 code	P2 code	P1 exec	P2 exec
A = 1	B = 1	issue "store A"	issue "store B"
read B	read A	issue "load B"	issue "load A"
		commit A , send inv (winner)	flush at load A
			commit B, send inv

- SC can be maintained, but expensive, so may also use TSO or PC
  - Speculative execution and rollback can still improve performance
- Performance on contemporary multiprocessors: ILP + Strong MC ≅ Weak MC