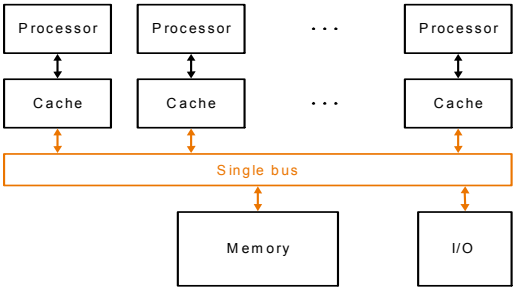
Cache coherence in SMPs

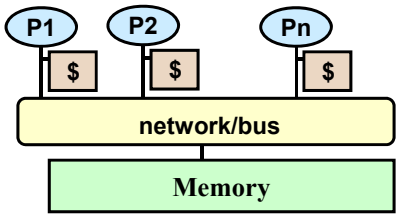
- Different caches may contain different value for the same memory location.



Time	Event	Cache Contents for CPU A	Cache Contents for CPU B	Memory Contents for location X
0				1
1	CPU A Reads X	X = 1		1
2	CPU B reads X	X = 1	X = 1	1
3	CPU A stores 0 into X	X = 0	X = 1	0 if write through 1 if write back

Memory consistency models (§ 5.6)

- The issue of cache coherence is related to the memory consistency model.
- Intuitively, reading an address should return the last value written to that address (an ambiguous definition in multi-processors/multi-cores)



- **Sequential consistency**: "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Note: The system provides sequential consistency if every node of the system sees the effect of the memory (write) operations in the same order, which may be different from the order of issuing the operations in real time.

Example:

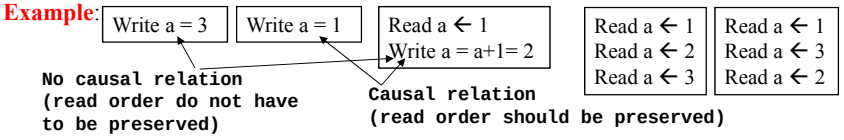
Write a = 2 Write b = 1 Write a = 10 Read a	Read a Read b Read a	…. Write a = 5 …..	Read a Read b Read a

Other consistency models

Strict consistency is a model which is more restrictive than sequential consistency. It demands that operations are seen in the order in which they were actually issued in real time, which is essentially impossible to secure in a distributed system.

Causal consistency is a model which is weaker than sequential consistency. It demands that read and write operations that are causally related are seen by every node of the system in the same order. However, concurrent writes may be seen in different order by different nodes (sequential consistency requires that all nodes see all writes in the same order.)

Example: when a node performs a read followed by a write to location B, the two operations are causally related if the written value depends on the read value.



Implications of sequential consistency

- Result of execution should be the same as long as:
  - Accesses on each processor were kept in order
  - Accesses on different processors were arbitrarily interleaved

- **Example:**

Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

- Should be impossible for both if-statements to be evaluated as true

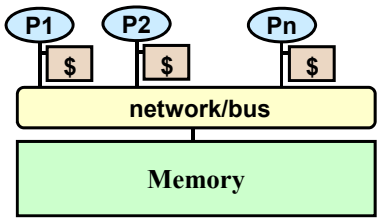
Sequential Consistency?

- A lot of hardware and technology to ensure cache coherence
- But the sequential consistency model may be broken anyway
  - The compiler reorders/removes code
  - Prefetching instructions causes reordering
  - The network may reorder two write messages

- Virtually all commercial systems give up on the idea of maintaining strong sequential consistency
  - Cache coherence enforces weaker memory models than Sequential Consistency
    - Writes to the **same location** by different processors are seen in the same order by all processors
  - Programmers can avoid race conditions by relying on coherence only across synchronization primitives (memory fence synchronization to flush caches – synchronization barriers)

Approaches to cache coherence

- Do not cache shared data
- Do not cache writeable shared data
- Explicitly flush caches when coherence is needed
- Use snoopy caches (if connected by a bus)
- If no shared bus, then
  - Use broadcast to emulate shared bus
  - Use directory-based protocols (to communicate only with concerned parties, not with everybody – directory keeps track of concerned parties)



An Example Snoopy Protocol (§ 5.2)

- Invalidation protocol for write-back caches
- Each block of memory can be:
  - Uncached: not in any caches
  - Clean in one or more caches and up-to-date in memory, or
  - Dirty in exactly one cache, or
- Correspondingly, we record the sate of each block in a cache as:
  - Invalid : block contains no valid data,
  - Shared : a clean block (can be shared by other caches), or
  - Modified/Exclusive: a dirty block (cannot be in any other cache)

MSI protocol = Modified/shared/invalid

Makes sure that if a block is dirty in one cache, it is not valid in any other cache and that a read request gets the most updated data

An Example Snoopy Protocol (cont.)

- A read miss to a block in a cache, C1, generates a bus transaction -- if another cache,C2, has the block "modified", it has to write back the block before memory supplies it. C1 gets the data from the bus and the block becomes "shared" in both caches.
- A write hit to a shared block in C1 forces an "Invalidate"-- Other caches that have the block should invalidate it– the block becomes "modified" in C1.
- A write hit to a modified block does not generate "Invalidate" or change of state.
- A write miss (to an invalid block) in C1 generates a bus transaction
  - If a cache, C2, has the block as "shared", it invalidates it
  - If a cache, C2, has the block in "modified", it writes back the block and changes it state in C2 to "invalid".
  - If no cache supplies the block, the memory will supply it.
  - When C1 gets the block, it sets its state to "modified"

Example

- Assume that blocks B1 and B2 map to the same cache location L.
- Initially neither B1 or B2 is cached
- Block size = one word

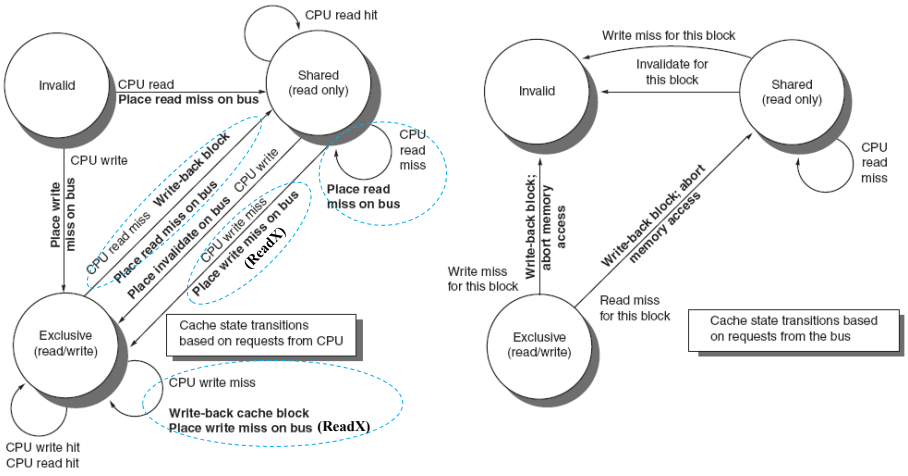
Event	In P1's cache	In P2's cache
	L = invalid	L = invalid
P1 writes 10 to B1 (write miss)	L ← B1 = 10 (modified)	L = invalid
P1 reads B1 (read hit)	L ← B1 = 10 (modified)	L = invalid
P2 reads B1 (read miss)	B1 is written back L ← B1 = 10 (shared)	L ← B1 = 10 (shared)
P2 writes 20 to B1 (write hit)	L = invalid	Put invalidate B1 on bus L ← B1 = 20 (modified)
P2 writes 40 to B2 (write miss)	L = invalid	B1 is written back L ← B2 = 40 (modified)
P1 reads B1 (read miss)	L ← B1 = 20 (shared)	L ← B2 = 40 (modified)

Example (cont.)

Event	In P1's cache	In P2's cache
	L ← B1 = 20 (shared)	L ← B2 = 40 (modified)
P1 writes 30 to B1 (write hit)	Put Invalidate B1 on bus L ← B1 = 30 (modified)	L ← B2 = 40 (modified)
P2 writes 50 to B1 (write miss)	B1 is written back L ← invalid	B2 is written back L ← B1 = 50 (modified)
P1 reads B1 (read miss)	L ← B1 = 50 (shared)	B1 is written back L ← B1 = 50 (shared)
P2 reads B2 (read miss)	L ← B1 = 50 (shared)	L ← B2 = 40 (shared)
P1 writes 60 to B2 (write miss)	L ← B2 = 60 (modified)	L = invalid

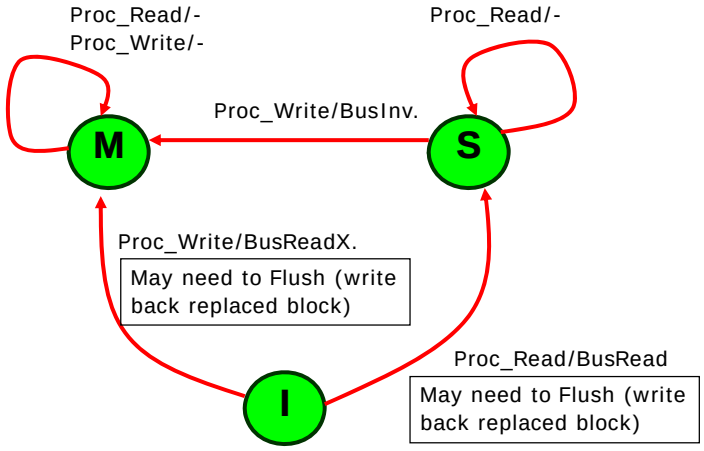
Describe the messages that show up on the bus after each event
Message types: read request, read to write request (ReadX),
write back (Flush), invalidate

Formal specification

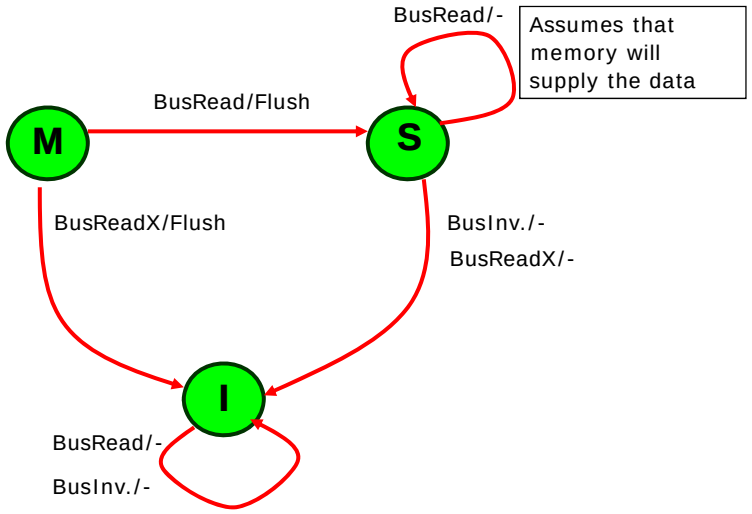


Note: Miss transactions enclosed in ⬭ indicate that the cache contains a block that is not the requested block (a block that should be evicted).

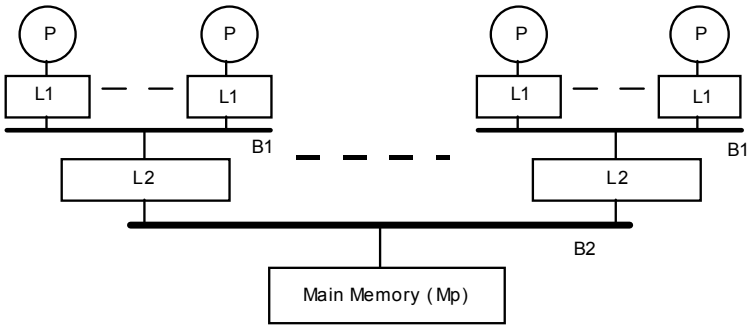
State Transition Diagram
Processor Initiated Transactions



State Transition Diagram
Bus Initiated Transactions

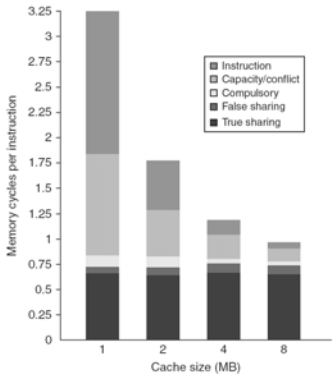


Hierarchical Snooping



- B1 busses follow standard snooping protocol to keep L1 copies consistent with the content of its shared L2
- The B2 bus keeps copies in L2 consistent with memory
- A change of state of a cache line in L2 can be triggered by a transaction on one of the two busses (B1 or B2) and may require propagation to the other bus.
- Is the inclusion property needed??

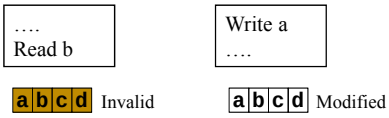
Performance study example (§ 5.3)



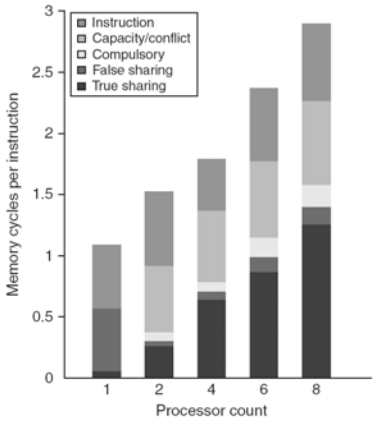
CPI for commercial benchmarks

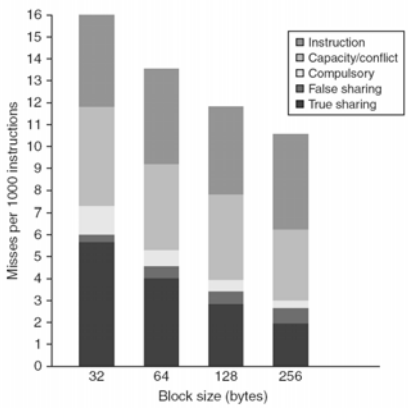
Coherence misses:

- True sharing misses
  - Write to a shared block
  - Read an invalid block
- False sharing misses
  - Read an unmodified word in an invalidated block



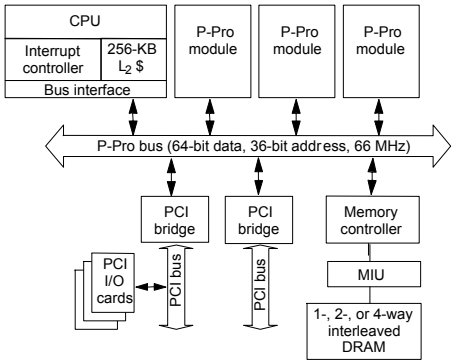
Performance study example





How do you handle coherence if you do not have a shared bus??

Sample Machines



- Sun Enterprise server
  - Coherent
  - Up to 16 processor and/or memory-I/O cards

- Intel Pentium Pro Quad
  - Coherent
  - 4 processors