Computer Architecture

计 算 机 体 系 结 构

第17讲：TLP（3）

张献伟

xianweiz.github.io

DCS3013, 11/30/2022

Review Questions

- data coherence issue?

  Processors may see different values of the same data.
- coherence vs consistency?

  Cache vs memory, what vs when, same vs different locations
- why let hardware enforce coherence?

  More efficient; lower programming burden
- two classes of protocols?

  Snooping, directory.
- snooping coherence protocol?

  Each core tracks sharing status of each block.
- write invalidation?

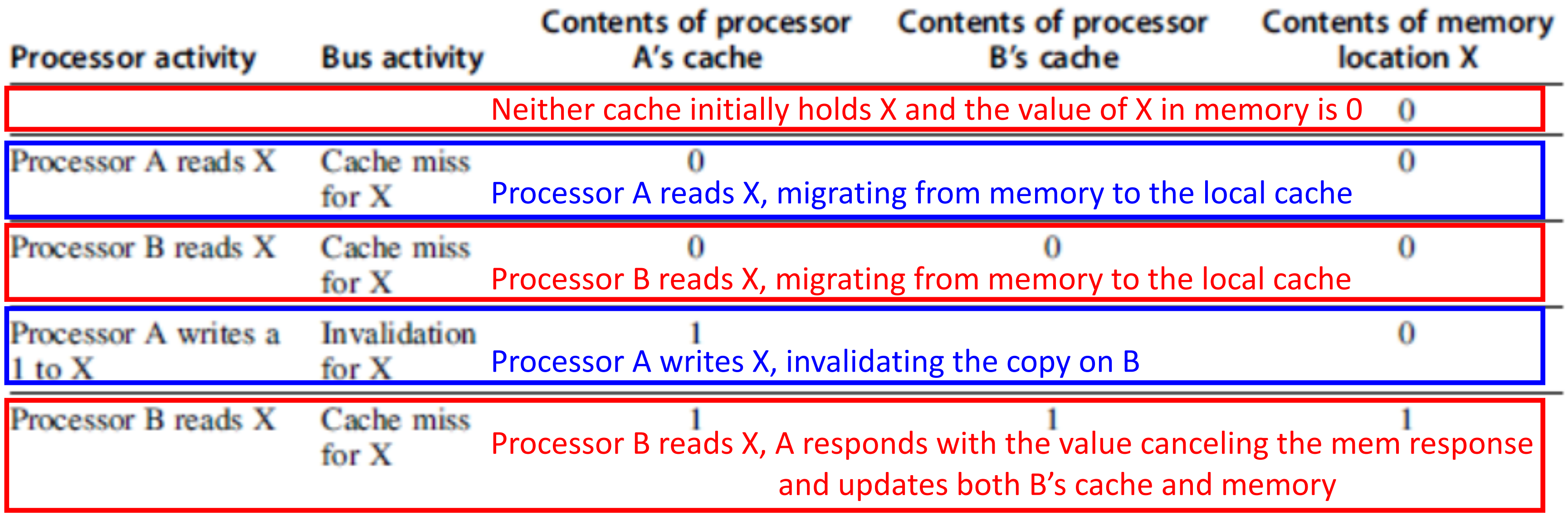
  Invalidate all other copies before writing.

Snooping Coherence Protocols[窥探]

- Write invalidation protocol[写无效]
  - Ensure that a processor has <u>exclusive access</u> to a data item before it writes that item
  - Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs
    - All other cached copies of the item are **invalidated** (👉 that's the name)
- Write update/broadcast protocol[写更新]
  - Update all the cached copies of data item when that item is written
  - Must broadcast all writes to shared cache lines, and thus consumes considerably more bandwidth
- Write invalidation protocol is by far the most common
  - We'll focus on it

Write Invalidation Protocol[写无效]

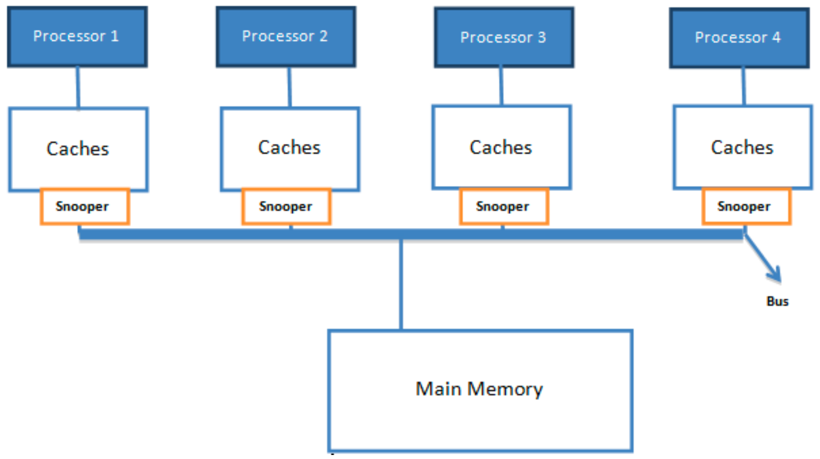
- Write invalidate
  - On write, invalidate all other copies
  - Use bus itself to serialize
- Example
  - Invalidation protocol working on a snooping bus for a single block (X) with write-back caches



Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
		Neither cache initially holds X and the value of X in memory is 0		0
Processor A reads X	Cache miss for X	0		0
		Processor A reads X, migrating from memory to the local cache		
Processor B reads X	Cache miss for X	0	0	0
		Processor B reads X, migrating from memory to the local cache		
Processor A writes a 1 to X	Invalidation for X	1		0
		Processor A writes X, invalidating the copy on B		
Processor B reads X	Cache miss for X	1	1	1
		Processor B reads X, A responds with the value canceling the mem response and updates both B's cache and memory		

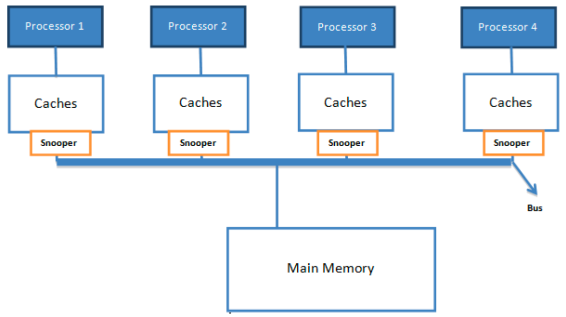
Snoopy Implementation[窥探实现]

- Key is to use bus, or another broadcast medium, to perform invalidates
- To perform an invalidate
  - The processor simply acquires bus access and broadcasts the address to be invalidated on the bus[获得总线，广播地址]
  - All processors continuously snoop on the bus, watching the addresses[窥探总线，收听地址]
  - The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated[核对地址，作废数据]



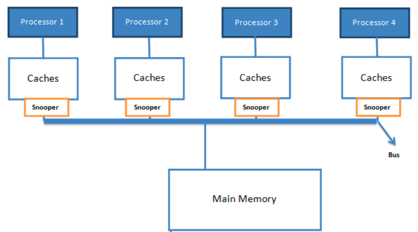
Snoopy Implementation (cont.)

- When a <u>write</u> to a block that is <u>shared</u> occurs,[要写共享块]
  - The writing processor must acquire bus access to broadcast its invalidation
- If <u>two processors</u> attempt to <u>write shared blocks</u> at the same time,[两个处理器想同时写到共享块]
  - Their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus[串行‘无效’操作]
  - The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated[作废数据]
  - If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes[串行写操作]



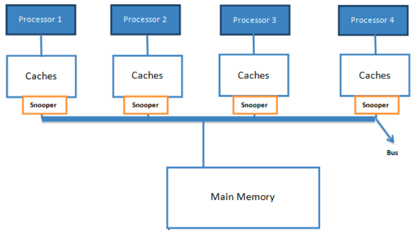
Snoopy Implementation (cont.)

- Locate a data item when a cache <u>miss</u> occurs,[找到数据]
  - For write-through cache, easy to find the recent value[写通]
    - All written data are always sent to the memory
  - For write-back cache, harder to find the most recent value[写回]
    - The newest value can be in a private cache rather than in the shared cache or memory
- Happily, write-back caches can use the same snooping scheme both for cache <u>misses</u> and for <u>writes</u>[同样窥探]
  - Each processor snoops every address placed on the shared bus[每个处理器窥探每个地址]
  - If a processor finds that it has a dirty copy of the requested cache block, it provides that block in response to the read request and causes the memory (or lower-level cache) access to be aborted[某个处理器拥有脏数据→ 响应]



Snoopy Implementation (cont.)

- Normal cache tags can be used to implement snooping, and the valid bit for each block makes invalidation easy to implement
  - Read misses, whether generated by an invalidation or by other events, are simply relying on the snooping capability
  - For writes, we'd like to know whether any other copies of the block are cached, because[是否独一份？]
    - If no other copies, then the write need not be placed on the bus
- Add an extra bit to track whether a block is shared
  - The bit is used to decide whether a write must generate an invalidate
    - Write to shared: invalidate, then mark block as "exclusive"
    - Sole copy of a cache block is normally called "owner"



SUN YAT-SEN UNIVERSITY 中山大學

MSI Protocol

- Invalidation protocol for write-back caches
- Each data block can be[数据块状态]
  - Uncached: not in any cache
  - Clean in one or more caches and up-to-date in memory, or
  - Dirty in exactly one cache

Dirty in more caches???

- Correspondingly, we record the coherence state of each block in a cache as[一致性状态]
  - 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

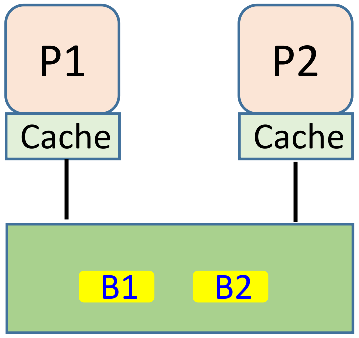
https://people.cs.pitt.edu/~melhem/courses/2410p/ch5-2.pdf

MSI Protocol (cont.)

- A read miss to a block in a cache, *C1*, generates a bus transaction[读不命中]
  - If another cache, *C2*, has the block “modified”, it has to write back the block before memory supplies it[其他cache有新数据]
    - *C1* gets data from the bus and the block becomes “shared” in both caches
- A write hit to a shared block in *C1* forces an “Invalidate”[写命中-’共享’]
  - Other caches that have the block should invalidate it – the block then becomes “modified” in *C1*[其他cache作废数据]
- A write hit to a modified block does not generate “Invalidate” or change of state[写命中-’修改’]
- A write miss (to an invalid block) in *C1* generates a bus transaction[写不命中]
  - 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



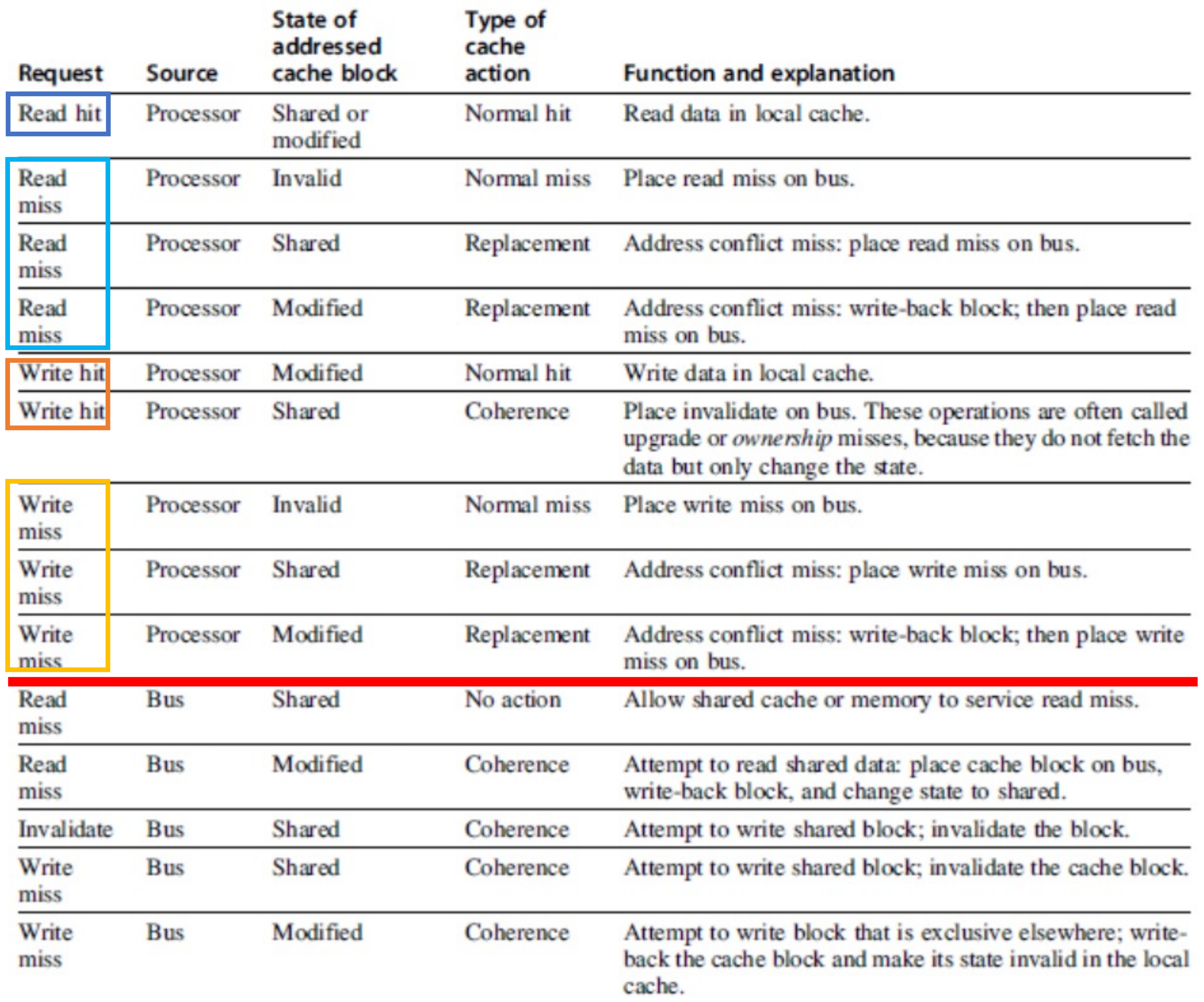
Event	In P1's cache	In P2's cache
	L = invalid	**L = invalid**
P1 writes 10 to B1 (write miss)	L <- B1 = 10 (modified)	L = invalid
P1 reads B1 (read hit)	L <- B1 = 10 (modified)	L = invalid
P2 reads B1 (read miss)	B1 is written back L <- B1 = 10 (shared)	L <- B1 = 10 (shared)
P2 writes 20 to B1 (write 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.)

- When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the block invalidate it
- For a write miss, if the block is exclusive in just one private cache, that cache also writes back the block
  - Otherwise, the data can be read from the shared cache or memory

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

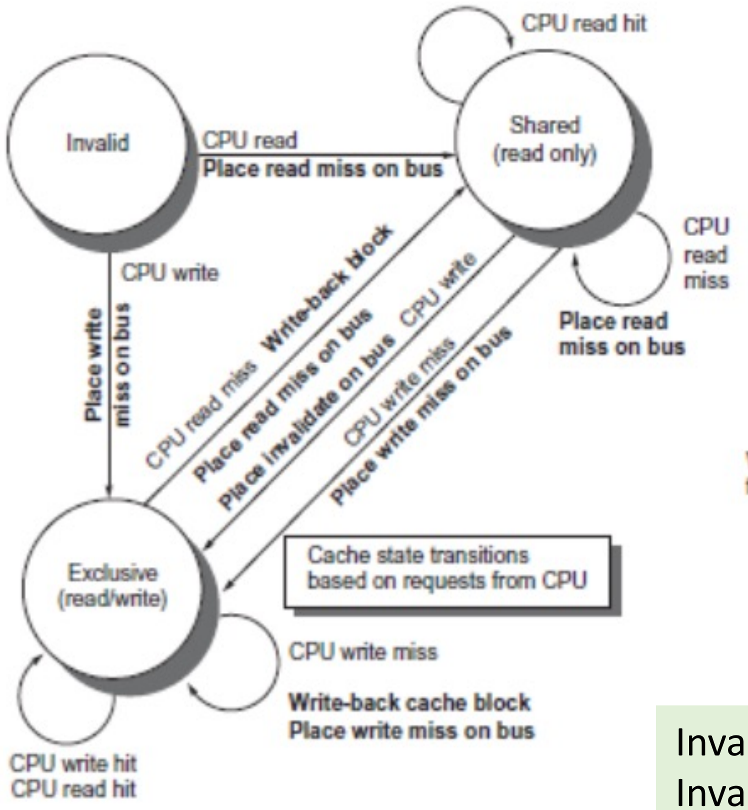
The Protocol

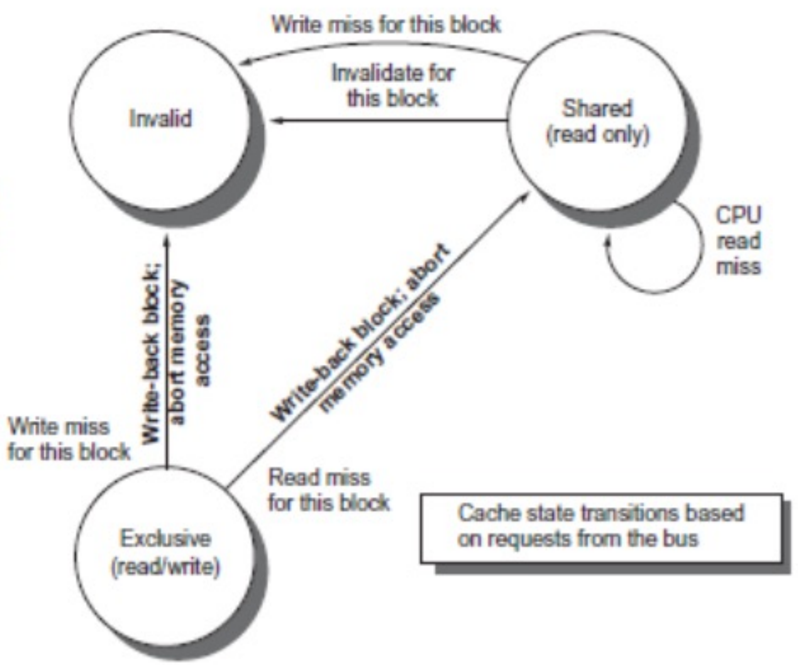


Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Formal Specification[形式化定义]

- Finite state transition diagram for a single private cache block[状态转换图]
  - Transitions based on processor and bus requests, respectively





Write miss for this block
Invalidate for this block
Invalid
Shared (read only)
CPU read miss
Write-back block; abort memory access
Write-back block; abort memory access
Write miss for this block
Read miss for this block
Exclusive (read/write)
Cache state transitions based on requests from the bus

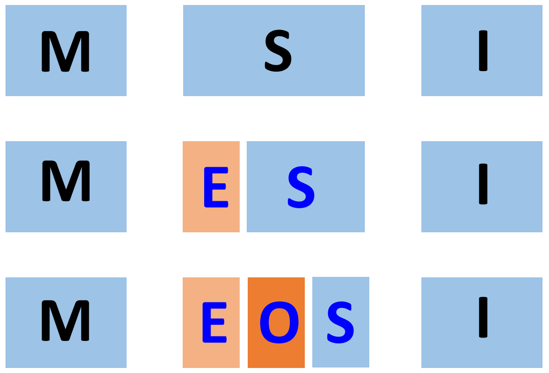
Invalid/Exclusive → Shared: a read happens
Invalid/Shared → Exclusive: a write happens
Shared/Exclusive → Invalid: write-invalidation

MSI Issues & Extensions[扩展]

- Complications for the basic MSI protocol
  - Operations are not atomic[非原子操作]
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
  - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- MSI: always invalidate before writing

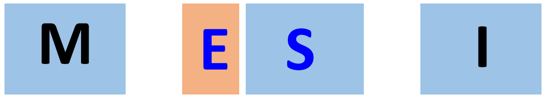
Is it necessary?

- Extensions
  - Adding additional states and transitions, which optimize certain behaviors, possibly resulting in improved performance
  - Two common extensions
    - **MESI**: new ‘Exclusive’
    - **MOESI**: new ‘Exclusive’ and ‘Owner’

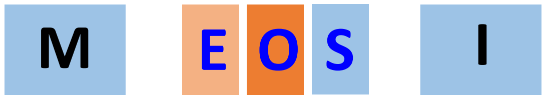


MESI and MOESI

- MESI adds state Exclusive



  - Shared: Exclusive (only one cache) + Shared (2 or more caches)
  - Indicate when a cache block is resident <u>only in a single cache</u> but is <u>clean</u>[其他cache都没有]
  - A subsequent write to a block in *E* state by the same core need not acquire bus access or generate an invalidate
- MOESI further adds state Owner



  - Shared: Shared Modified (O) + Shared Clean (S)
  - Indicate that the associated block is <u>owned by that cache</u> and <u>out-of-date in memory</u>[独有，且比内存新]
  - In MSI/MESI, when sharing a block in *M* state, the state is changed to *S*, and the block must be written back to memory
  - In MOESI, the block can be changed from *M* to *O* without writing it to memory

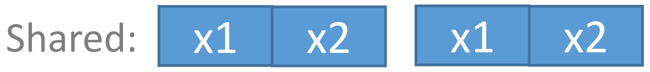
https://people.engr.ncsu.edu/efg/506/sum99/001/lec9-coherence.pdf

Performance of SMPs: Misses

- In a multicore using a snooping coherence protocol, overall cache performance is a combination of
  - The behavior of uniprocessor cache miss traffic
  - The traffic caused by communication, resulting in invalidations and subsequent cache misses
- Three C’s classification of uniprocessor misses
  - Capacity(容量), compulsory(冷启动), conflict(地址冲突)
- Coherence misses caused by interprocessor communication[一致性缺失]
  - **True sharing misses**: directly arise from the sharing of data among processors
  - **False sharing misses**: the miss would not occur if the block size were a single word

Performance of SMPs: Misses (cont.)

- **True sharing misses**, in an invalidation-based protocol
  - The first write by a processor to a shared block causes an invalidation to establish ownership of that block (invalidate all)
  - When another processor tries to read a modified word in that block, a miss occurs and the resultant block is transferred (invalidated by the store)
- **False sharing misses**
  - Caused by the coherence alg. with a single valid bit per block
  - Occurs when a block is invalidated (and a subsequent reference causes a miss)
    - Some word in the block, other than the one being read, is written into



Time	P1	P2
1	Write x1	
2		Read x2

False sharing miss: x2 was invalidated by 'write x1' in P1