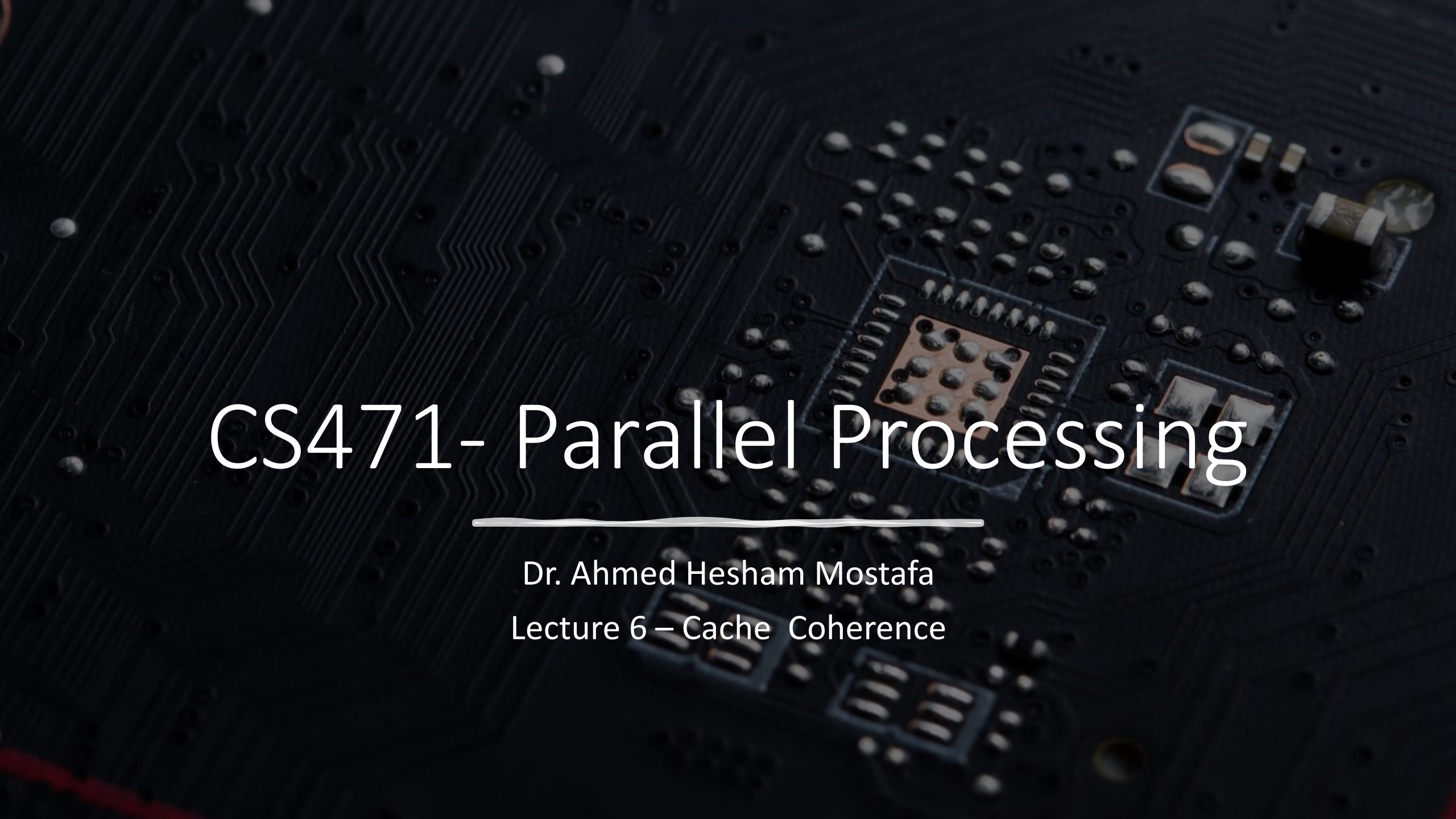


CS471- Parallel Processing

Dr. Ahmed Hesham Mostafa

Lecture 6 – Cache Coherence

Cache Coherence

- The primary advantage of cache is its ability to reduce the average access time in uniprocessors.
- When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer.
- If the operation is to write, there are two transfer. If the operation is to write, there are two commonly used procedures to update memory.
  - write-through policy
  - write-back policy

Cache Coherence

- **Write through**: All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.
- In the **write-through** policy, both cache and main memory are updated with every write operation.
- In the **write-back** policy, only the cache is updated, and the location is marked so that it can be copied later into main memory.
- **Write back**: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.

Cache Coherence

- In a shared memory multiprocessor system, all the processors share a common memory.
- In addition, each processor may have a local memory, part or all of which may be a cache.
- The same information may reside in a number of copies in some caches and main memory.
- To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical.

Cache Coherence

- This requirement imposes a cache coherence problem.
- A memory scheme is coherent **if the value returned on a load instruction is always the value given by the latest store instruction with the same address**.
- Without a proper solution to the cache coherence problem, caching cannot be used in bus-oriented multiprocessors with two or more processors.

Conditions for Incoherence

- Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data.
- Read-only data can safely be replicated without cache coherence mechanisms.
- To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12.
- Sometime during the operation an element X from main memory is loaded into the three processors, P1, P2, and P3.
- As a consequence, it is also copied into the private caches of the three processors.
- For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory.

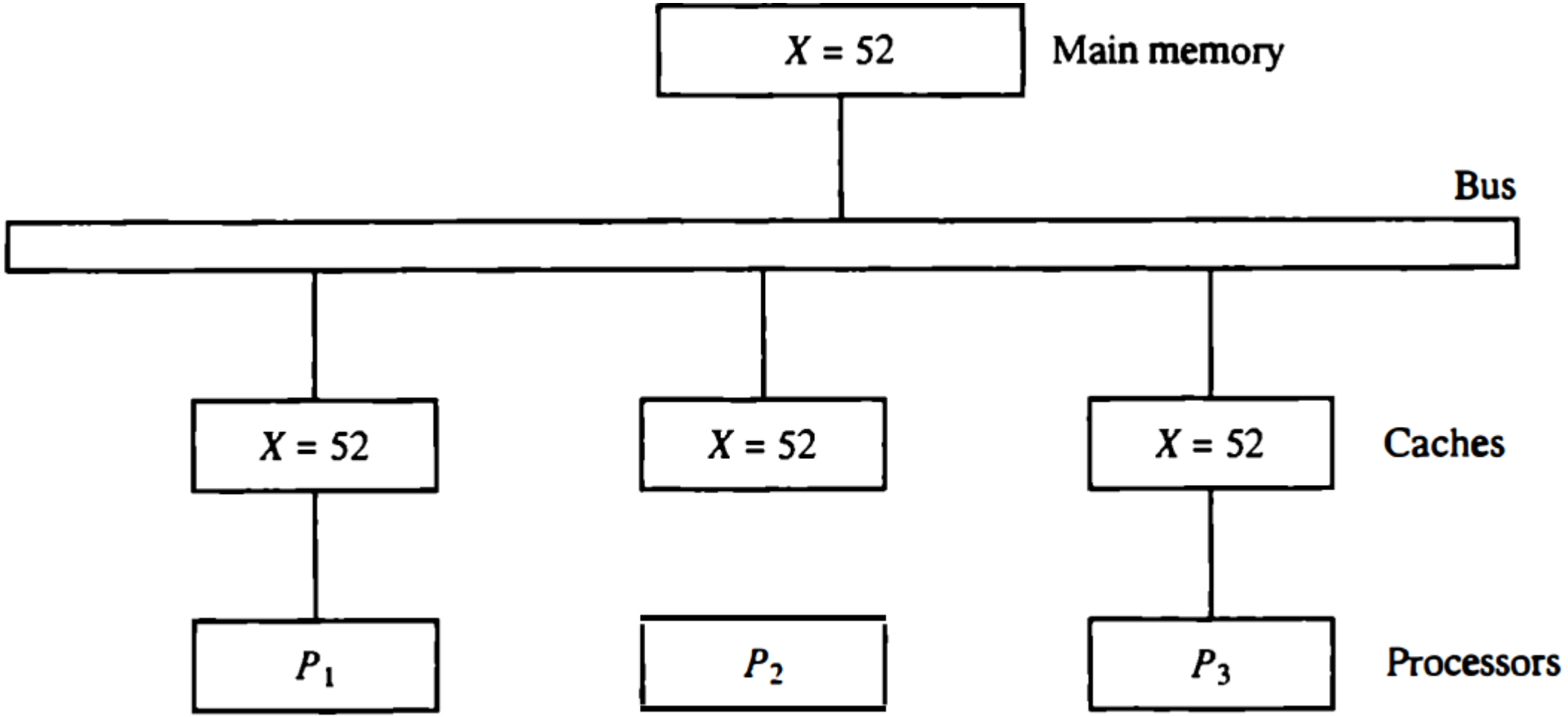
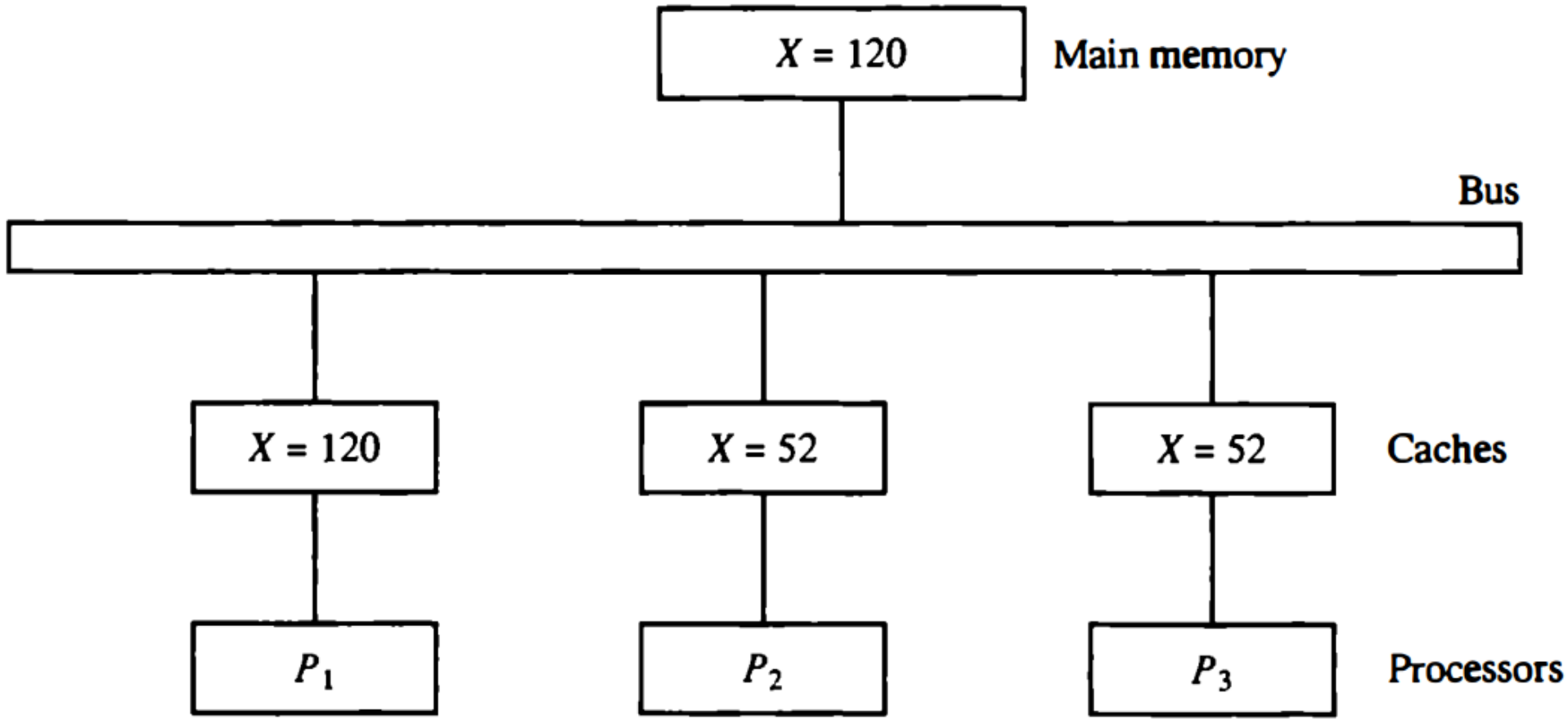


Figure 13-12 Cache configuration after a load on X.

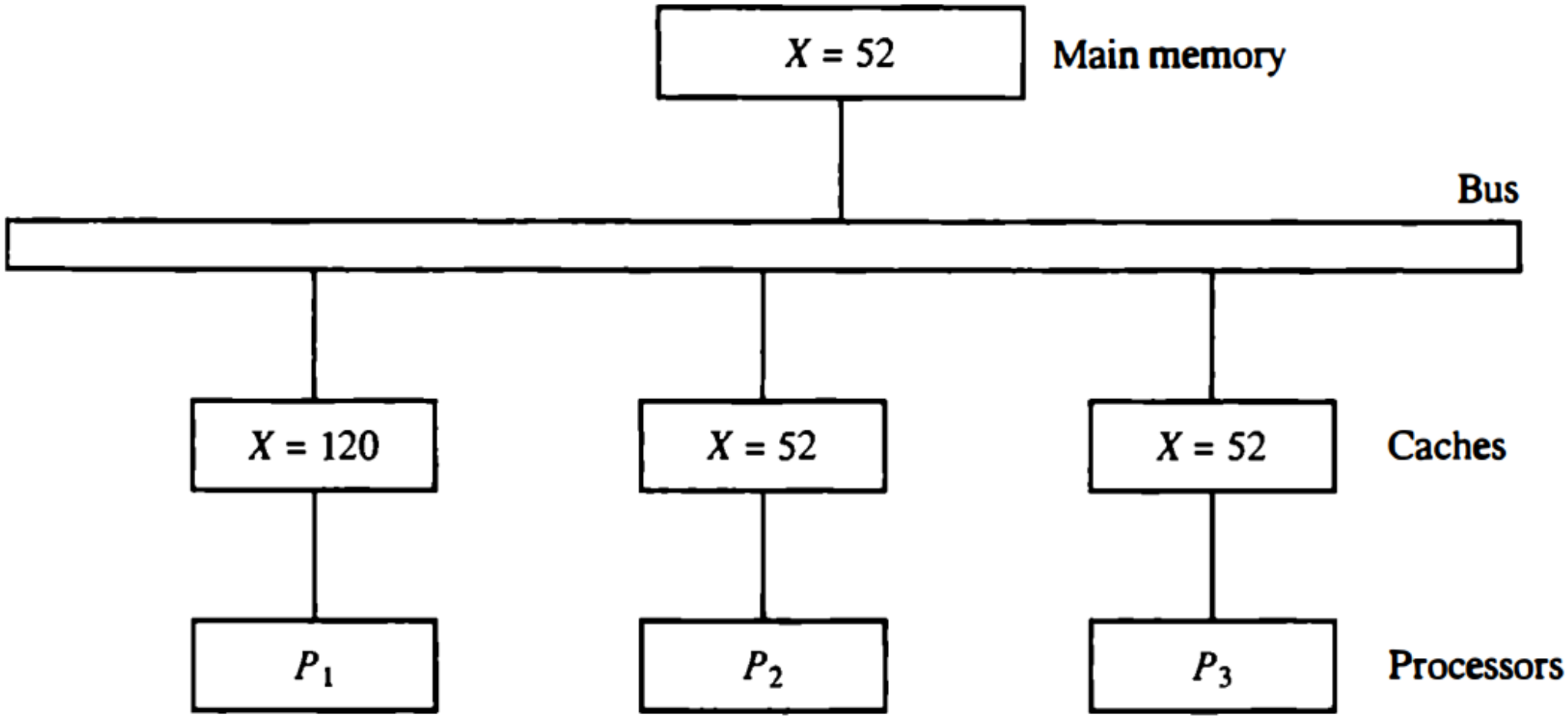
Conditions for Incoherence

- If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value.
- Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to' the cache.
- This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P1 updates memory to the new value in a write-through policy.

Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy



(b) With write-back cache policy

Conditions for Incoherence

- In write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value.
- In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated when the modified data in the cache are copied back into memory.

Solutions to the Cache Coherence Problem

- A First scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory.
- Every data access is made to the shared cache.
- This method violates the principle of closeness of Cpu to cache and increases the average memory access time.
- In effect, this scheme solves the problem by avoiding it.

Solutions to the Cache Coherence Problem

- Second scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called Cachable.
- Shared writable data are noncachable.
- The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches.
- The noncachable data remain in main memory.
- This method restricts the type of data stored in caches and introduces an extra software overhead that may affect the performance.

Solutions to the Cache Coherence Problem

- Third scheme that allows writable data to exist in at least one cache is a method that employs a centralized global table in its compiler.
- The status of memory blocks is stored in the central global table.
- Each block is identified is identified as read-only (RO) or read and write (RW).
- All caches can have copies of blocks identified as RO.
- Only one cache can have a copy of an RW block.
- Thus if the data are updated in the cache with RW block, the other caches are not affected because they do not have a copy of this block.

Solutions to the Cache Coherence Problem

- The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes.
- The two methods mentioned in previous slides use software-based procedures require the ability to tag information in order to disable caching of shared writable data.
- Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency.

Solutions to the Cache Coherence Problem

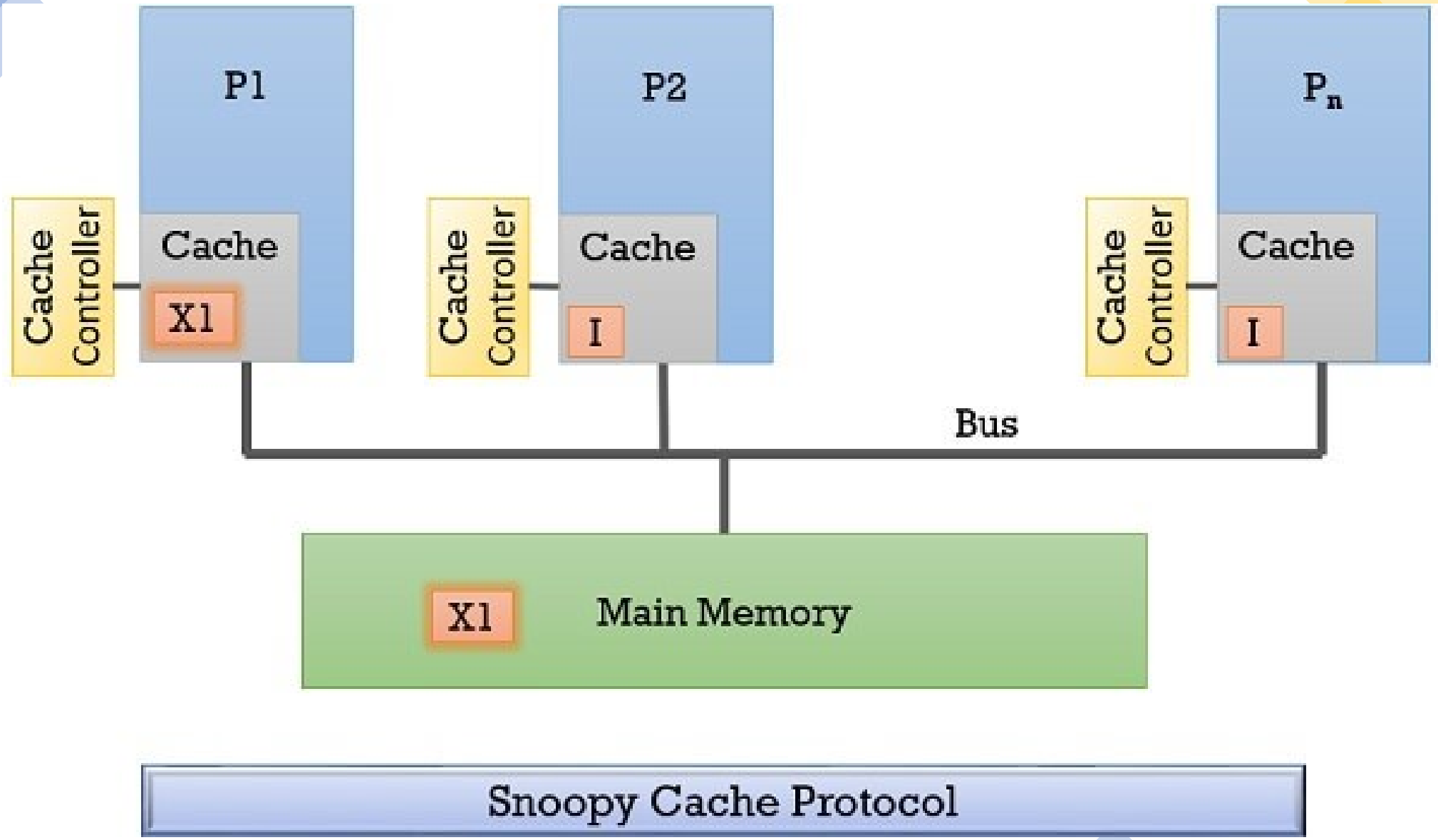
- In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs.
- All caches attached to the bus constantly monitor the network for possible write operations.
- Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected.
- The bus controller that monitors this action is referred to as a snoopy cache controller.
- This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

Snoopy Protocol

- In the multiprocessor environment, all the processors are connected to memory modules via a **single bus**.
- The transaction between the processors and the memory module i.e. read, write, invalidate request for the data block occurs via bus.
- If we implement the **cache controller** to every processor's cache in the system, it will **snoop** all the **transactions over the bus** and perform the appropriate action. So, we can say that the Snoopy protocol is the **hardware solution** to the cache coherence problem.
- It is used for small multiprocessor environments as the large shared-memory multiprocessors are connected via the interconnection network.

Snoopy Protocol

- Consider a scenario from write-back, if a processor has just modified a data block in its cache, and is a current owner of the block.
- Now, if processor P1 wishes to modify the same data block that has been modified. P1 would broadcast the invalidation request on the bus and becomes the owner for that data block and modify the data block. The other processors who have the copy of the same data block **snoop** the bus and invalidate their copy of the data block (I). It updates memory using the write-back protocol.


P1
P2
P_n
Cache Controller
Cache
X1
Cache Controller
Cache
I
Cache Controller
Cache
I
Bus
X1
Main Memory
Snoopy Cache Protocol

MESI Protocol

- For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:
- **Modified(M)**: The line in the cache has been modified (different from main memory) and is available only in this cache.
- **Exclusive(E)**: The line in the cache is the same as that in main memory and is not present in any other cache.
- **Shared(S)**: The line in the cache is the same as that in main memory and may be present in another cache.
- **Invalid(I)**: The line in the cache does not contain valid data.

MESI Protocol

- write hit means the content on which you want to write is present in cache,
- write miss is when content on which write to be performed is not in cache.
- read hit means the content on which you want to read is present in cache,
- read miss is when content on which read to be performed is not in cache.

MESI Protocol

- In the MESI (Modified, Exclusive, Shared, Invalid) cache coherence protocol, when a read miss occurs and the data is available in another cache, the protocol allows for the data to be obtained from the cache that currently holds a copy of the data.
- This is known as a "cache-to-cache transfer" or "cache-to-cache copy."

MESI Protocol

- When a cache block changes its status from M, it first updates the main memory.

Event	*Local* Its Cache	*Remote* others Cache
Read hit	Use local copy	No action
Read miss	I to S, or I to E	(S,E,M) to S
Write hit	(S,E) to M	(S,E,M) to I
Write miss	I to M	(S,E,M) to I

MESI Protocol

- **Read Miss in a Cache (e.g., Cache A):** When a cache experiences a read miss and the required data is not present in the cache (Invalid state), it needs to obtain the data from somewhere else.
- **Check Other Caches:** The MESI protocol allows caches to communicate with each other to maintain coherence. If another cache (e.g., Cache B) has the required data in the Shared or Exclusive state, Cache A can request the data from Cache B.

MESI Protocol

- **Cache-to-Cache Transfer:** If Cache B has the data and is in the Shared state, it can send the data directly to Cache A. Cache A updates its own copy of the data and continues with the read operation.
- **Avoiding Main Memory Access:** By allowing cache-to-cache transfers, the MESI protocol helps avoid unnecessary accesses to the main memory (Memory access is abandoned).
- This can improve overall system performance by reducing memory latency.

MESI Local Read Miss (1)

- No other copy in caches
  - – Processor makes bus request to memory
  - – Value read to local cache, marked E
- •If Another cache has E copy
  - – Processor makes bus request to memory
  - – Snooping cache puts copy value on the bus
  - – Memory access is abandoned
  - – Local processor caches value (the local cache get the value)
  - – Both lines set to S

MESI Local Read Miss (2)

- Several caches have S copy
  - – Processor makes bus request to memory
  - – One cache puts copy value on the bus
- (arbitrated)
  - – Memory access is abandoned
  - – Local processor caches value
  - – Local copy set to S
  - – Other copies remain S

MESI Local Read Miss (3)

- One cache has M copy
  - – Processor makes bus request to memory
  - – Snooping cache puts copy value on the bus
  - – Memory access is abandoned
  - – Local processor caches value
  - – Local copy tagged S
  - – Source (M) value copied back to memory
  - – Source value M -> S

MESI Local Write Hit (1)

- Line must be one of MES
- • M
  - – line is exclusive and already ‘dirty’
  - – Update local cache value
  - – no state change
- • E
  - – Update local cache value
  - – State E -> M

MESI Local Write Hit (2)

- S
  - – Processor broadcasts an invalidate on bus
  - – Snooping processors with S copy change S->I
  - – Local cache value is updated
  - – Local state change S->M

MESI Local Write Miss (1)

- Detailed action depends on copies in other processors
- • No other copies
  - – Value read from memory to local cache
  - – Value updated
  - – Local copy state set to M

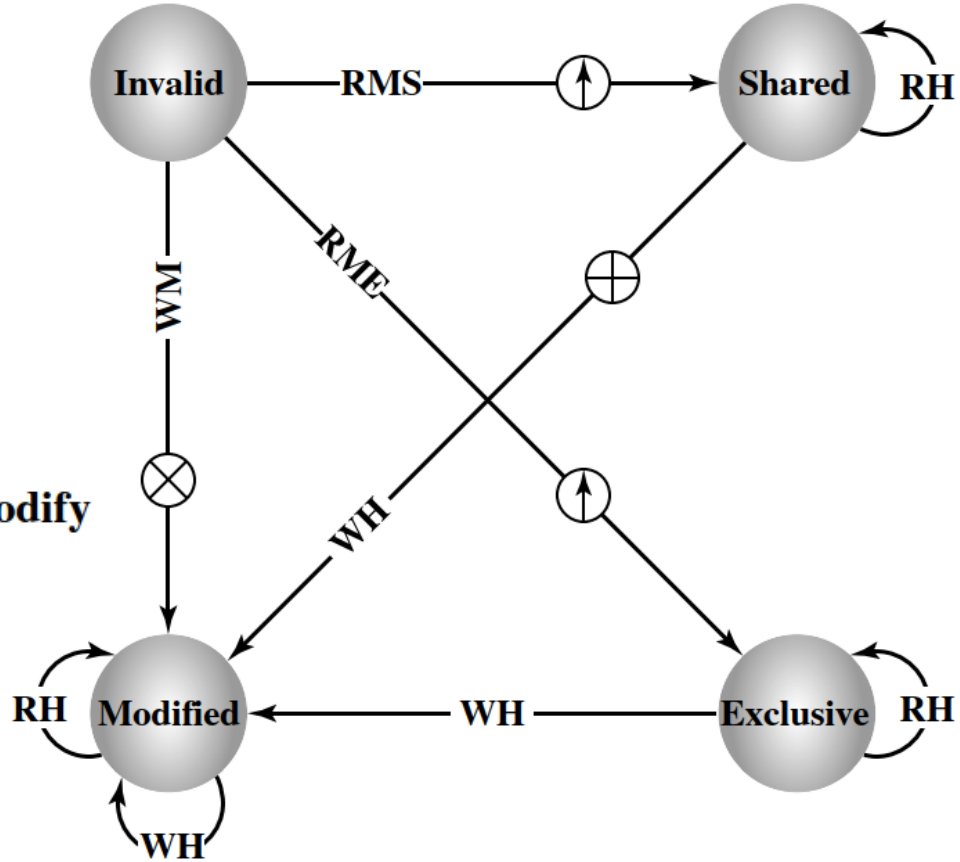
MESI Local Write Miss (2)

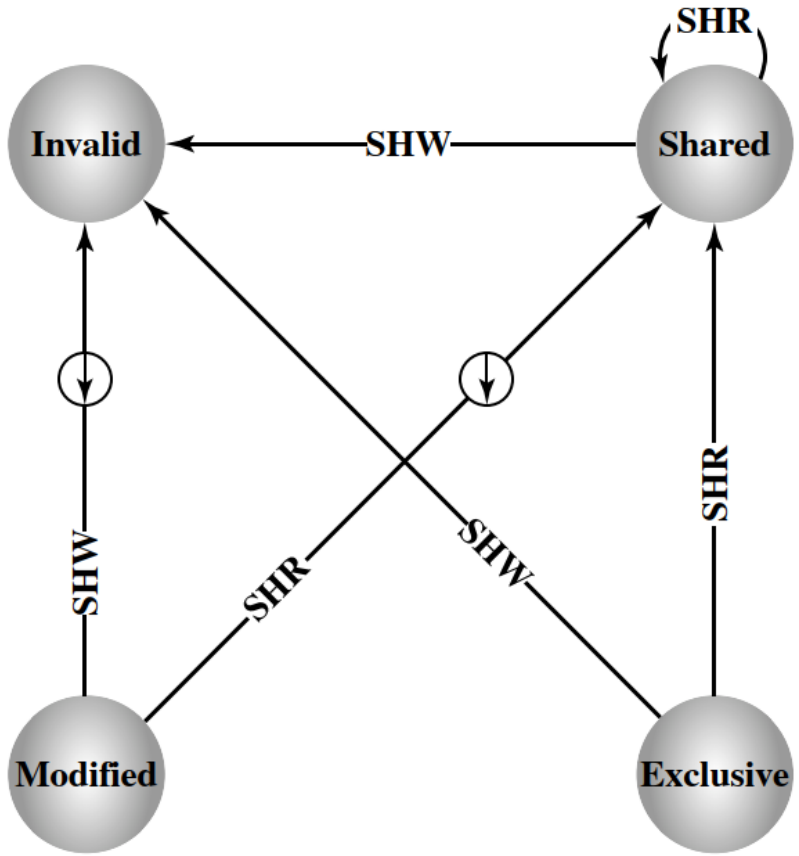
- Other copies, either one in state E or more in state S
- – Value read from memory to local cache - bus
- transaction marked RWITM (read with intent to modify)
- – Snooping processors see this and set their copy state to I
- – Local copy updated & state set to M

MESI Local Write Miss (3)

- Another copy in state M
  - • Processor issues bus transaction marked RWITM
  - • Snooping processor sees this
  - – Blocks RWITM request
  - – Takes control of bus
  - – Writes back its copy to memory
  - – Sets its copy state to I

RH	**Read hit**
RMS	**Read miss, shared**
RME	**Read miss, exclusive**
WH	**Write hit**
WM	**Write miss**
SHR	**Snoop hit on read**
SHW	**Snoop hit on write or read-with-intent-to-modify**





Event	*Local* Its Cache	*Remote* others Cache
Read hit	Use local copy	No action
Read miss	I to S, or I to E	(S,E,M) to S
Write hit	(S,E) to M	(S,E,M) to I
Write miss	I to M	(S,E,M) to I

Dirty line copyback

Invalidate transaction

Read-with-intent-to-modify

Cache line fill

MESI Protocol Example 1

- A four-processor shared-memory system implements MESI protocol
- For the following sequence of memory references,
  show the state of the line containing the variable X in each processor's cache after each reference is resolved
- All processors start out with the line containing X invalid in their cache

	P0's state	P1's state	P2's state	P3's state
Initial State	*I*	*I*	*I*	*I*
P0 reads X	*E*	*I*	*I*	*I*
P1 reads X	*S*	*S*	*I*	*I*
P2 reads X	*S*	*S*	*S*	*I*
P3 writes X	*I*	*I*	*I*	*M*
P0 reads X	*S*	*I*	*I*	*S*

MESI Protocol Example 2

- This is a bus-based shared memoty multiprocessor system with 2 CPUs, MESI Protocol and write-back caches. Both CPUs access the shared variables B and C.

Instruction	P0 cash-Var B	P1 cash-Var B	P0 cash-Var C	P1 cash-Var C
Initial	I	I	I	I
P0 read B	E	I	I	I
P1 read C	E	I	I	E
P0 read C	E	I	S	S
P1 write C	E	I	I	M
P0 Write B	M	I	I	M

References

Thanks