Chapter 5. Assorted Topics on Scalable Architectures

Contents

Reference : *Kai Hwang's book - Chapter 7 + class slides as indicated*

Pages: 331-336 (before sect 7.1.2), 345 - 347; 348-350;

Read the slides for Snoopy protocol; Reading assignment : 358-359(before full-map directories);

Cache Coherence Problem

The problem of data inconsistency between the MM and cache is referred to as the *cache coherence problem*

Three possible sources:

- Sharing of data that can be modified
- Process migration
- I/O activity

Refer to Figs. 7.12, 7.13 on pages 349, 350

Two protocols are suggested and followed widely in practice

- Snoopy Bus Protocols
- Directory Based Protocols

Snoopy Bus protocol:

We have a bus watcher unit with each processor that observes the transactions on the bus.

In particular, the watcher monitors all the mem write operations.

Two different approaches are possible. These are

(a). Write-invalidate protocol
(b). Write-update protocol

Further, the basic protocol in achieving the consistency can be through:

(i). Write-through caches
(ii). Write-back caches

Write-through caches:

We will consider only the *write-invalidate protocol.*

For write-through caches, we have two possible states that a cache block can assume whenever a *read, write, and a replacement* operations are effected : *valid and invalid* states

In valid state - all processors can read safely
In invalid state - block is invalidated or being replaced

Whenever a processor writes into its local cache, all other cache copies become invalidated.

Write-back caches: Here,

valid state -> RW (read-write) state
+ RO (read only) state

invalid state - same as before

This scheme corresponds to an *ownership* protocol.

Ownership Protocol

- In this scheme, whenever a SM (shared memory) owns a block, the remote caches can only contain RO copies of the block.

- This means that multiple copies can exist, but all the processors are allowed to only read.

- The RW state corresponds to only one copy existing in the entire system owned by the local processor, say i. Then, read and write can be performed safely.

Modifying a block using Ownership Protocol

Suppose a block needs to be modified by a processor, then:

- First it has to obtain a permission to do so. This is done by broadcasting a request to all caches and the SM.
- <u>If a modified block exists</u> in a remote cache, then the SM is first updated and all other caches will be invalidated.
- After updating, the ownership is transferred to the requesting cache.

Remarks: *Write invalidate* vs *Write Update*

- A write invalidate protocol may lead to a heavy bus traffic caused by read-misses, resulting from the processor updating of a variable and other processors trying to read the same variable.
- Also, the write update protocol may update data items in remote caches which will never be used by other processors.

These problems pose additional limitations in using buses to build large multiprocessors.

Consequently, we have....

Directory based protocols :

Snoopy bus methods are not suitable when the number of processors is large, as broadcasting becomes more expensive.

This is an alternative protocol which recommends every node to maintain a directory of the blocks that are currently cached.

Two flavours :

Full Map directory method and *Limited Directories.*

Read pages358 and 359 (before Full-Map directories)

Workings of Directory-based protocol

System: Shared memory; N Processors with each with its local cache; higher bus bandwidth (scalability w.r.t N is possible);

A *directory* is used to ensure cache coherence.

Main purpose of the directory residing in the main memory: To know which blocks are in caches and their status.

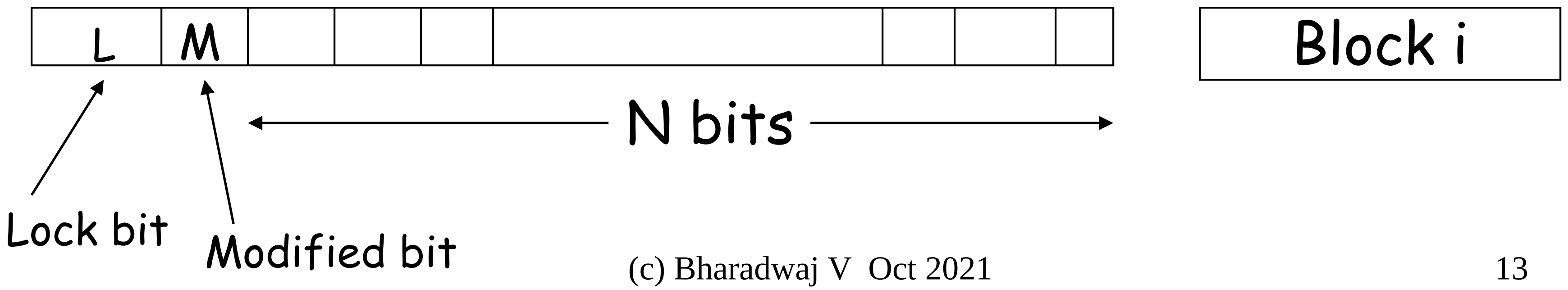
Suppose a Multi-Pro has M blocks in main memory and there are N processors and each processor has a cache.

Each memory block has an N bit directory.

Consider the following example which demonstrates the working of directory-based protocol.

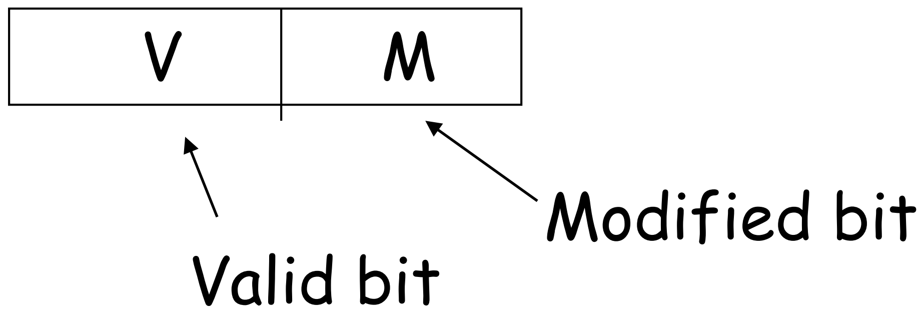
Number of Main Memory blocks: M
Number of Processors: N

Each MM block has N bit directory entry as follows.
Space complexity: O(MN) -> expensive!



If kth processor's cache has this block i, then the kth bit is set to 1

Similarly for each block in cache memory we maintain a two-bit information as follows.



Following table identifies different states that can exist in the system for a block

For a MM block

State	L	M	Remarks
Absent (A)	0	0	No cache holds a copy of this block
Present (P)	1	0	One or more caches has this block
Present Ex (PM)	0	1	Exactly one cache has this copy
Locked (L)	1	1	Operation on this block is going on and it is locked

For a Cache Memory block

State	V	M	Remarks
Invalid (I)	0	X	Contents of this block on cache are invalid
Valid (V)	1	0	Valid and unmodified
Valid M (VM)	1	1	Valid, modified, and it is the only copy.

Note: Cache consistency is maintained by actions taken by mem controllers. The actions depend on read/write requests, state of the block in cache and main memory.

Now consider how we use these tables while performing the following operation.

Scenario: "Read from Memory to Processor Register'

Assumptions:

- Write-invalidate protocol
- Write-through cache

Description: kth processor executes "load" instructio

Case I: Read hit

This means that a block containing a word is in the kth processor's cache

a) if status of the block in cache is V or VM, then read the word; no other action.

b) Suppose if the status is I, then read the valid copy from the cache of a processor holding a valid copy

Case II: Read Miss

This means that the block is not in the local cache. We have to check the status of this block in the main memory

a) If the status of the block is *A*, then we do the following.

(i) Replacement algorithm to be invoked and bring the block from MM to the k-th processor's cache

(ii) Full-fill the read request from cache (write-through cache)

(iii) Enter a '1' in the k-th bit position of the directory entry of this block

(iv) Change the status bits of the directory entry of this block to **PM**

(v) Change the status bits of the k-th processor cache block to **V**

(b) If the status of the block in MM is P, then carry out the actions mentioned above - (i), (ii), (iii), and (v)

(c) If the status of the block in MM is PM, then carry out the actions - (i), (ii), and (iii) and the following.

(vi) Change the status bits of the directory entry of this block to **P**

(v) Change the status bits of k^{th} processor's cache block to **V**

Hot spot problem in shared memory systems

We can divide shared-memory multiprocessors into:

(i) uniform memory access (UMA) systems
(ii) non-uniform memory access (NUMA) systems.

In UMA multiprocessors, the cost of accessing a memory location by any processor in the system is the same.

In NUMA multiprocessors, the shared memory is physically distributed among the processors. Thus, some memory locations are closer to a processor and less expensive to access than others resulting in a non-uniform memory access cost.

In a UMA multiprocessor the shared memory is global to all processors. An interconnection network facilitates communication between the processors and the global shared memory. Typically, UMA MPs use a single bus as the interconnection network.

In such systems one of the common problems is memory contention.

Note that memory contention occurs even if these requests are to different data items, all located in the same memory module. Such a memory module is called a "hot memory module" and the phenomenon is called hot-spot contention.

To alleviate this problem, it was suggested that multiple requests can be combined under a conflicting situation.

An atomic *primitive*, referred to as

Fetch&Add(x,e)

has been developed to perform parallel memory updates using the combining network

Refer to the Fig. 7.11 on page 346

```
Fetch&Add (x,e)  /* x: integer variable stored
{                      in the SM; e: increment
  temp  <---    x;
      x  <---  temp + e;
  return temp;
} /* atomic operation */
```

The idea is based on a *serialization* principle. The values returned to the nodes are unique, depending on the order in which this addition is followed. On the whole, the result is equal to performing N additions, but performed in one *indivisible* operation.

Handling parallel executions of N independent iterations can also be effected using this primitive.

```
Doall N =1 to 100
   {....}
Endall
```

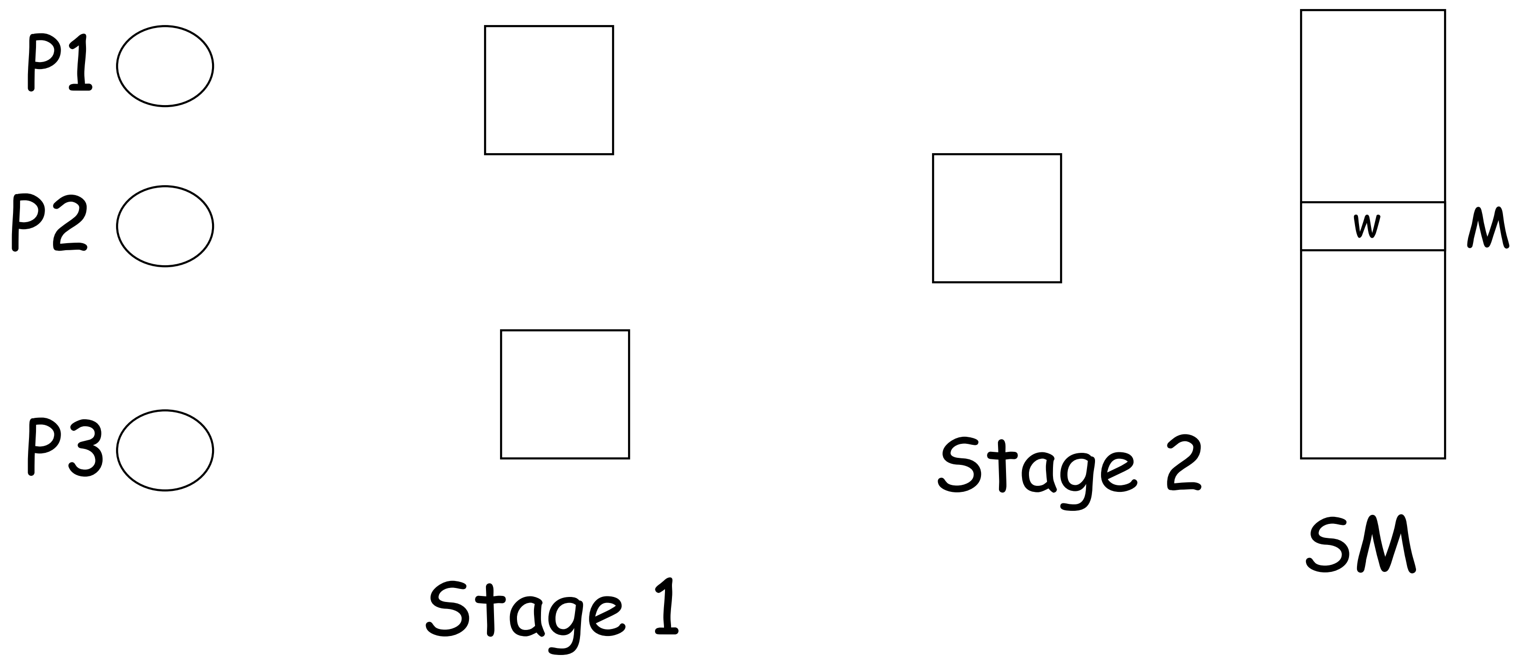
If each processor executes the following code, a unique value of N is returned to each processor. This is useful when the system tries to control the issuance of work to the available processors.

```
i <-- Fetch&Add(N,1)  /* N being initialized as 1
While (i <= 100) Doall

        { ....code using i...}

i <-- Fetch&Add (N,1)
Endall
```

The use of such combined networks is certainly costly, however, with faster and cheaper technology in the future we can optimize the cost.

Try the following :

Using the 2 stage network shown below, show how the following set of F&A operations are carried out in that order.



Order of execution ↓

Processor 1: F&A (M, x)

Processor 2: F&A (M, y)

Processor 3: F&A (M, z)

M is a memory location

Assume that the location M has a value w stored initially. Assume that the switches at the stages have buffers to store.

Hierarchical bus systems

Catch the link "Notes on Hierarchical Bus Systems" (Chapter 5) – *Reading Assignment*