Shared Memory Architecture

A shared memory computer system consists of a set of independent processors, a set of memory modules, and an interconnection network as shown in Figure
1-All processors share a global memory (coordination and synchronization).
2-Communication between tasks running on different processors is performed through writing to and reading from the global memory.

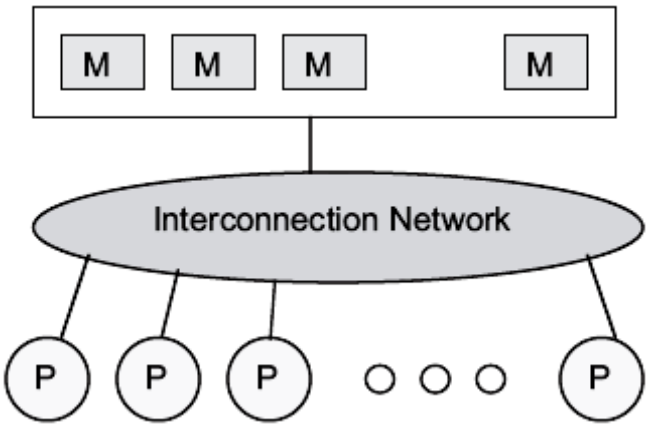

Figure -1 Shared memory systems.

Two main problems need to be addressed when designing a shared memory system: performance degradation due to contention, and coherence problems.

1- Classification of Shared Memory Systems

An arbitration unit within the memory module passes requests through to a memory controller. If the memory module is not busy and a single request arrives, then the arbitration unit passes that request to the memory controller and the request is satisfied. The module is placed in the busy state while a request is being serviced. The memory module sends a wait signal, through the memory controller, to the processor making the new request.

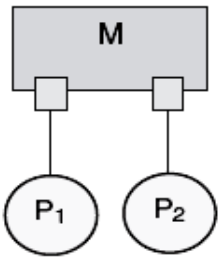

Figure -2 Shared memory via two ports.

Again, the denied request can be either held to be served next or it may be repeated some time later. Based on the interconnection network used, shared memory systems can be categorized in the following categories.

1.1- Uniform Memory Access (UMA)

The interconnection network used in the UMA can be a single bus, multiple buses, or a crossbar switch. Because access to shared memory is balanced, these systems are also called SMP (symmetric multiprocessor) systems. Each processor has equal opportunity to read/write to memory, including equal access speed.

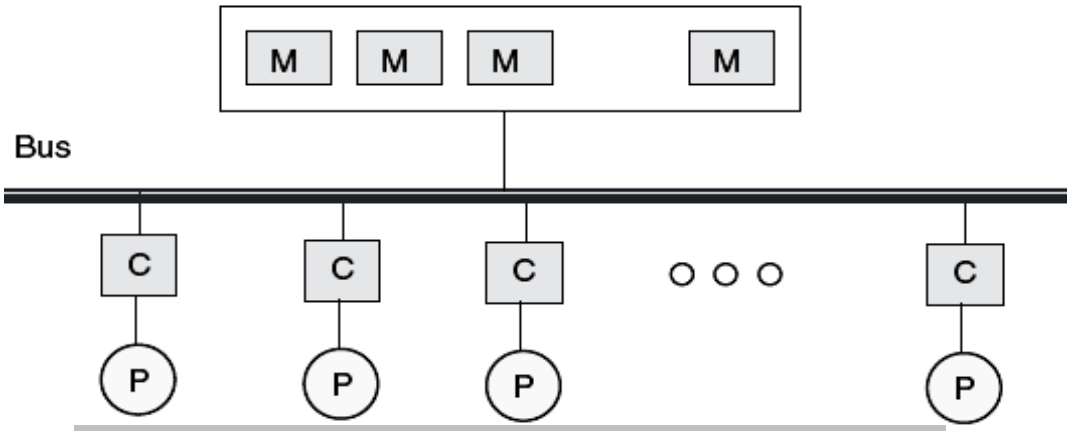

Figure -3 Bus-based UMA (SMP) shared memory system.

1.2- Nonuniform Memory Access (NUMA)

In the NUMA system, each processor has part of the shared memory attached. The memory has a single address space. Therefore, any processor could access any memory location directly using its **real address**. However, the access time to modules depends on the distance to the processor. This results in a nonuniform memory access time. A number of architectures are used to interconnect processors to memory modules in a NUMA. Among these are the tree and the hierarchical bus networks

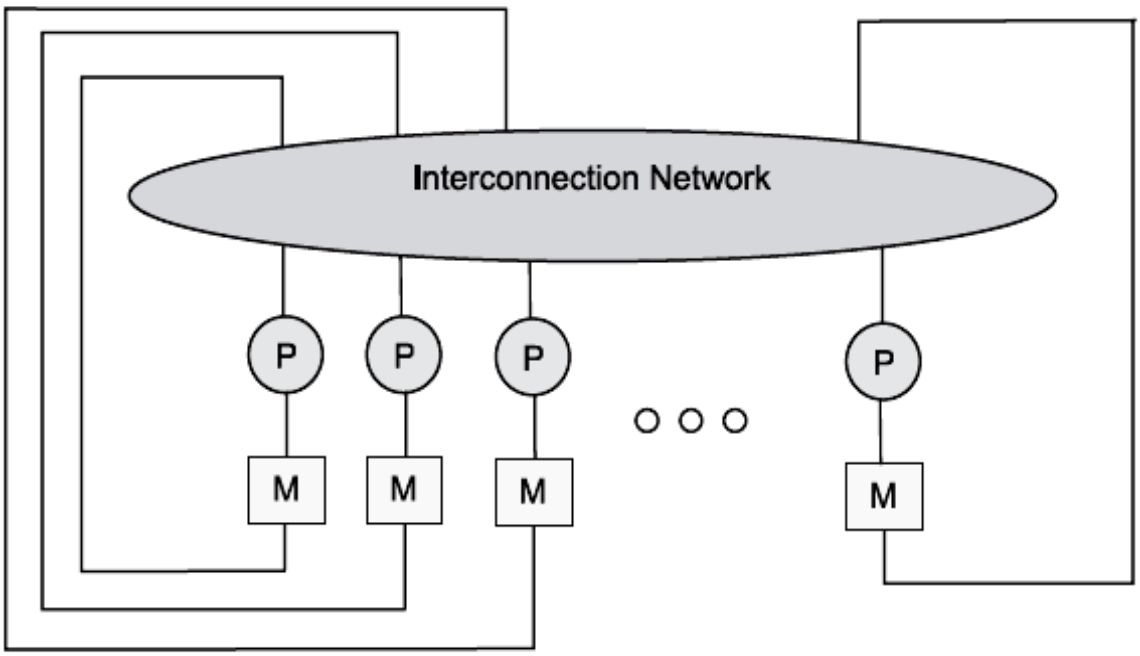

Figure -4 NUMA shared memory system.

1.3- Cache-Only Memory Architecture (COMA)

Similar to the NUMA, each processor has part of the shared memory in the COMA. However, in this case the shared memory consists of cache memory.. There is no memory hierarchy and the address space is made of all the caches. There is a cache directory (D) that helps in remote cache access.

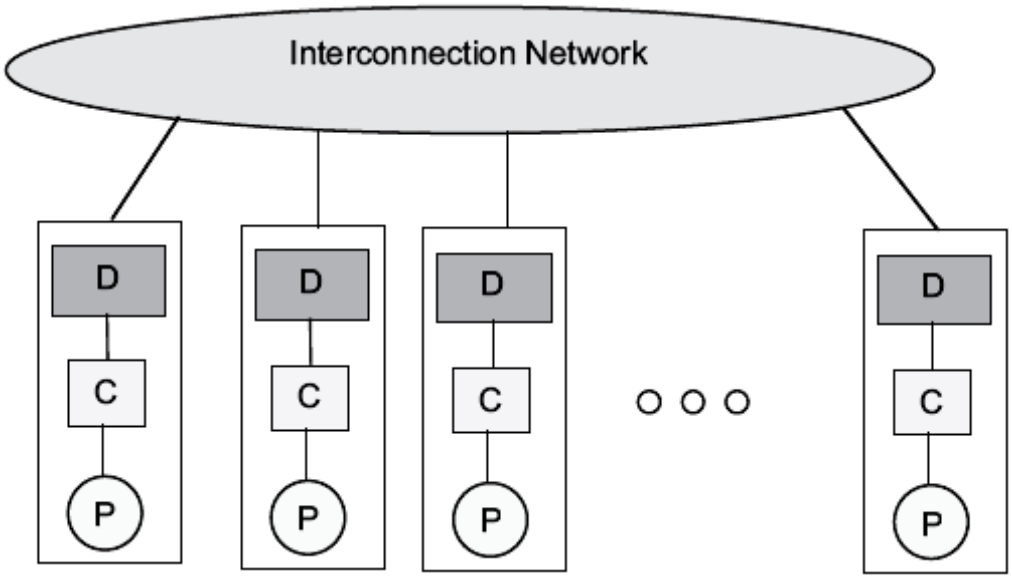

Figure -5 COMA shared memory system.

2- Bus-Based Symmetric Multiprocessors

The bus/cache architecture alleviates the need for expensive multiported memories and interface circuitry as well as the need to adopt a message-passing paradigm when developing application software. However, the bus may get saturated if multiple processors are trying to access the shared memory (via the bus) simultaneously.

Highspeed caches connected to each processor on one side and the bus on the other side mean that local copies of instructions and data can be supplied at the highest possible rate. If the local processor finds all of its instructions and data in the local cache, we say the hit rate is 100%. The miss rate of a cache is the fraction of the references that cannot be satisfied by the cache, and so must be copied from the global memory, across the bus, into the cache, and then passed on to the local processor.

Hit rates are determined by a number of factors, ranging from the application programs being run to the manner in which cache hardware is implemented.We want to minimize the number of times each local processor tries to use the central bus. Otherwise, processor speed will be limited by bus bandwidth.

We define the variables for hit rate, number of processors, processor speed, bus speed, and processor duty cycle rates as follows:

- N = number of processors;
- h = hit rate of each cache, assumed to be the same for all caches;

- (1 - h) = miss rate of all caches;
- B = bandwidth of the bus, measured in cycles/second;
- I = processor duty cycle, assumed to be identical for all processors, in fetches/cycle; and
- V = peak processor speed, in fetches/second.

The maximum number of processors with cache memories that the bus can support is given by the relation,

$$N \leq \frac{BI}{(1-h)V}$$

Example 1 Suppose a shared memory system is constructed from processors that can execute V = 107 instructions/s and the processor duty cycle I =1. The caches are designed to support a hit rate of 97%, and the bus supports a peak bandwidth of B = 106 cycles/s. Then, (1 - h) = 0.03, and the maximum number of processors N is N ≤ 106/(0.03 * 107) = 3.33. Thus, the system we have in mind can support only three processors!

We might ask what hit rate is needed to support a 30-processor system. In this case, h = 1- BI/NV = 1 - (106(1))/((30)(107)) = 1 - 1/300, so for the system we have in mind, h = 0.9967. Increasing h by 2.8% results in supporting a factor of ten more processors.

3- Basic Cache Coherency Methods

Multiple copies of data, spread throughout the caches, lead to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies. leading to incorrect final results.

3.1- Cache–Memory Coherence

In a single cache system, coherence between memory and the cache is maintained using one of two policies: (1) write-through, and (2) write-back. When a task running on a processor P requests the data in memory location X, for example, the contents of X are copied to the cache, where it is passed on to P. When P updates the value of X in the cache, the other copy in memory also needs to be updated in order to maintain consistency. In write-through, the memory is updated every time the cache is updated, while in write-back, the memory is updated only when the block in the cache is being replaced. TABLE -1 shows the write-through versus write-back policies.

TABLE -1 Write-Through vs. Write-Back

		Write-Through		Write-Back	
Serial	Event	Memory	Cache	Memory	Cache
1		X		X	
2	P reads X	X	X	X	X
3	P updates X	X′	X′	X	X′

3.2- Cache–Cache Coherence

In multiprocessing system, when a task running on processor P requests the data in global memory location X, for example, the contents of X are copied to processor P's local cache, where it is passed on to P. Now, suppose processor Q also accesses X. What happens if Q wants to write a new value over the old value of X?

There are two fundamental cache coherence policies: (1) write-invalidate, and (2) write-update. Write-invalidate maintains consistency by reading from local caches until a write occurs. When any processor updates the value of X through a write, posting a dirty bit for X invalidates all other copies. For example, processor Q invalidates all other copies of X when it writes a new value into its cache. This sets the dirty bit for X. Q can continue to change X without further notifications to other caches because Q has the only valid copy of X. However, when processor P wants to read X, it must wait until X is updated and the dirty bit is cleared. Write-update maintains consistency by immediately updating all copies in all caches. All dirty bits are set during each write operation. After all copies have been updated, all dirty bits are cleared. TABLE -2 shows the write-update versus write-invalidate policies.

TABLE -2 Write-Update vs. Write-Invalidate

		Write-Update		Write-Invalidate	
Serial	Event	P's Cache	Q's Cache	P's Cache	Q's Cache
1	P reads X	X		X	
2	Q reads X	X	X	X	X
3	Q updates X	X′	X′	INV	X′
4	Q updates X′	X″	X″	INV	X″

3.3- Shared Memory System Coherence

The four combinations to maintain coherence among all caches and global memory are:

. Write-update and write-through;
. Write-update and write-back;
. Write-invalidate and write-through;
. Write-invalidate and write-back.

If we permit a write-update and write-through directly on global memory location X, the bus would start to get busy and ultimately all processors would

be idle while waiting for writes to complete. In write-update and write-back, only copies in all caches are updated. On the contrary, if the write is limited to the copy of X in cache Q, the caches become inconsistent on X. Setting the dirty bit prevents the spread of inconsistent values of X, but at some point, the inconsistent copies must be updated.

4- Snooping Protocols

Snooping protocols are based on watching bus activities and carry out the appropriate coherency commands when necessary. Global memory is moved in blocks, and each block has a state associated with it, which determines what happens to the entire contents of the block. The state of a block might change as a result of the operations Read-Miss, Read-Hit, Write-Miss, and Write-Hit. A cache miss means that the requested block is not in the cache or it is in the cache but has been invalidated.

Snooping protocols differ in whether they update or invalidate shared copies in remote caches in case of a write operation. They also differ as to where to obtain the new data in the case of a cache miss. In what follows we go over some examples of snooping protocols that maintain cache coherence.

4.1 Write-Invalidate and Write-Through

In this simple protocol the memory is always consistent with the most recently updated cache copy. Multiple processors can read block copies from main memory safely until one processor updates its copy. At this time, all cache copies are invalidated and the memory is updated to remain consistent. The block states and protocol are summarized in TABLE -3.

Example 2 Consider a bus-based shared memory with two processors P and Q as shown in Figure -6. Let us see how the cache coherence is maintained using Write- Invalidate Write-Through protocol. Assume that that X in memory was originally set to 5 and the following operations were performed in the order given:
(1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X;
(6) P updates X; (7) Q reads X.

TABLE -4 shows the contents of memory and the two caches after the execution of each operation when Write-Invalidate Write-Through was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

TABLE -3 Write-Invalidate Write-Through Protocol

State	**Description**
Valid [VALID]	The copy is consistent with global memory.
Invalid [INV]	The copy is inconsistent.
Event	**Actions**
Read-Hit	Use the local copy from the cache.
Read-Miss	Fetch a copy from global memory. Set the state of this copy to Valid.
Write-Hit	Perform the write locally. Broadcast an Invalid command to all caches.

	Update the global memory.
Write-Miss	Get a copy from global memory. Broadcast an invalid command to all caches. Update the global memory. Update the local copy and set its state to Valid.
Block replacement	Since memory is always consistent, no write-back is needed when a block is replaced.

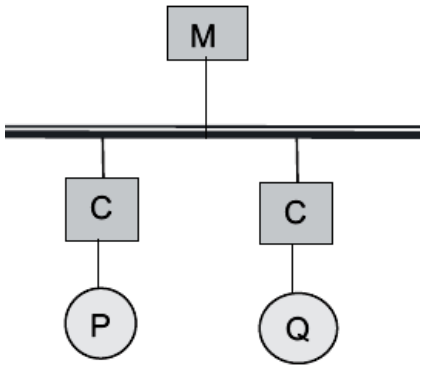

Figure -6 A bus-based shared memory system with two processors P and Q.

TABLE -4 Example 2 (Write-Invalidate Write-Through)

Serial	Event	Memory Location X	P's Cache		Q's Cache	
			Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VALID		
2	Q reads X (Read-Miss)	5	5	VALID	5	VALID
3	Q updates X (Write-Hit)	10	5	INV	10	VALID
4	Q reads X (Read-Hit)	10	5	INV	10	VALID
5	Q updates X (Write-Hit)	15	5	INV	15	VALID
6	P updates X (Write-Miss)	20	20	VALID	15	INV
7	Q reads X (Read-Miss)	20	20	VALID	20	VALID

4.2- Write-Invalidate and Write-Back (Ownership Protocol)

In this protocol a valid block can be owned by memory and shared in multiple caches that can contain only the shared copies of the block. Multiple processors can safely read these blocks from their caches until one processor updates its copy. At this time, the writer becomes the only owner of the valid block and all other copies are invalidated. The block states and protocol are summarized in TABLE -5.

TABLE -5 Write-Invalidate Write-Back Protocol

State	**Description**
Shared	Data is valid and can be read safely. Multiple copies can be in

(Read-Only) [RO]	this state.
Exclusive (Read-Write) [RW]	Only one valid cache copy exists and can be read from and written to safely. Copies in other caches are invalid.
Invalid [INV]	The copy is inconsistent.

Event	**Action**
Read-Hit	Use the local copy from the cache.
Read-Miss	If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (Read- Only) in both caches.
Write-Hit	If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).
Write-Miss	Get a copy from either a cache with an Exclusive (Read-Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).
Block replacement	If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write-back is needed when a block is replaced.

Example 3 Consider the shared memory system of Figure -6 and the following operations: (1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X; (6) P updates X; (7) Q reads X. TABLE -6 shows the contents of memory and the two caches after the execution of each operation when Write-Invalidate Write-Back was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

TABLE -6 Example 3 (Write-Invalidate Write-Back)

		Memory	P's Cache		Q's Cache	
Serial	Event	Location X	Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	RO		
2	Q reads X (Read-Miss)	5	5	RO	5	RO
3	Q updates X (Write-Hit)	5	5	INV	10	RW
4	Q reads X (Read-Hit)	5	5	INV	10	RW
5	Q updates X (Write-Hit)	5	5	INV	15	RW
6	P updates X (Write-Miss)	5	20	RW	15	INV
7	Q reads X (Read-Miss)	20	20	RO	20	RO

4.3- Write-Once

This write-invalidate protocol, which was proposed by Goodman in 1983, uses a combination of write-through and write-back. Write-through is used the very first time a block is written. Subsequent writes are performed using write-back. The block states and protocol are summarized in TABLE -7.

TABLE -7 Write-Once Protocol

State	**Description**
Invalid [INV]	The copy is inconsistent.
Valid [VALID]	The copy is consistent with global memory..
Reserved [RES]	Data have been written exactly once and the copy is consistent with global memory. There is only one copy of the global memory block in one local cache.
Dirty [DIRTY]	Data have been updated more than once and there is only one copy in one local cache. When a copy is dirty, it must be written back to global memory.

Event	**Action**
Read-Hit	Use the local copy from the cache.
Read-Miss	If no Dirty copy exists, then supply a copy from global memory. Set the state of this copy to Valid. If a dirty copy exists, make a copy from the cache that set the state to Dirty, update global memory and local cache with the copy. Set the state to VALID in both caches.
Write-Hit	If the copy is Dirty or Reserved, perform the write locally, and set the state to Dirty. If the state is Valid, then broadcast an Invalid command to all caches. Update the global memory and set the state to Reserved.
Write-Miss	Get a copy from either a cache with a Dirty copy or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Dirty.
Block replacement	If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid, Reserved, or Invalid states, no write-back is needed when a block is replaced.

Example 4 Consider the shared memory system of Figure -6 and the following operations: (1) P reads X; (2) Q reads X; (3) Q updates X; (4) Q reads X; (5) Q updates X; (6) P updates X; (7) Q reads X. TABLE -8 shows the contents of memory and the two caches after the execution of each operation when Write-Once was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

TABLE -8 Example 4 (Write-Once Protocol)

Serial	Event	Memory Location X	P's Cache		Q's Cache	
			Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VALID		
2	Q reads X (Read-Miss)	5	5	VALID	5	VALID
3	Q updates X (Write-Hit)	10	5	INV	10	RES
4	Q reads X (Read-Hit)	10	5	INV	10	RES
5	Q updates X (Write-Hit)	10	5	INV	15	DIRTY
6	P updates X (Write-Miss)	10	20	DIRTY	15	INV
7	Q reads X (Read-Miss)	20	20	VALID	20	VALID

4.4- Write-Update and Partial Write-Through

In this protocol an update to one cache is written to memory at the same time it is broadcast to other caches sharing the updated block. These caches snoop on the bus and perform updates to their local copies. There is also a special bus line, which is asserted to indicate that at least one other cache is sharing the block. The block states and protocol are summarized in TABLE -9.

TABLE -9 Write-Update Partial Write-Through Protocol

State	**Description**
Valid Exclusive [VAL-X]	This is the only cache copy and is consistent with global memory.
Shared [SHARE]	There are multiple cache copies shared. All copies are consistent with memory.
Dirty [DIRTY]	This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Copy ownership.)

Event	**Action**
Read-Hit	Use the local copy from the cache. State does not change.
Read-Miss	If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared in both caches. If the cache copy was in a Dirty state, the value must also be written to memory.
Write-Hit	Perform the write locally and set the state to Dirty. If the state is Shared, then broadcast data to memory and to all caches and set the state to Shared. If other caches no longer share the block, the state changes from Shared to Valid Exclusive.
Write-Miss	The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update and update all other caches that share the block and global memory. Set the state to Shared. If the copy comes from memory, perform the write and set the state to Dirty.

Block replacement	If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive or Shared states, no write-back is needed when a block is replaced.

Example 5 Consider the shared memory system of Figure -6 and the following operations: (1) P reads X; (2) P updates X; (3) Q reads X; (4) Q updates X; (5) Q reads X; (6) Block X is replaced in P's cache; (7) Q updates X; (8) P updates X. TABLE -10 shows the contents of memory and the two caches after the execution of each operation when Write-Update Partial Write-Through was used for cache coherence. The table also shows the state of the block containing X in P's cache and Q's cache.

TABLE -10 Example 5 (Write-Update Partial Write-Through)

Serial	Event	Memory Location X	P's Cache		Q's Cache	
			Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VAL-X		
2	P updates X (Write-Hit)	5	10	DIRTY		
3	Q reads X (Read-Miss)	10	10	SHARE	10	SHARE
4	Q updates X (Write-Hit)	15	15	SHARE	15	SHARE
5	Q reads X (Read-Hit)	15	15	SHARE	15	SHARE
6	Block X is replaced in P's cache (Replace)	15	–	–	15	VAL-X
7	Q updates X (Write-Hit)	15	–	–	20	DIRTY
8	P updates X (Write-Miss)	25	25	SHARE	25	SHARE

5- Directory Based Protocols

Owing to the nature of some interconnection networks and the size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical. For example, when a multistage network is used to build a large shared memory system, the broadcasting techniques used in the snoopy protocols becomes very expensive. In such situations, coherence commands need to be sent to only those caches that might be affected by an update. This is the idea behind directory-based protocols.

Cache coherence protocols that somehow store information on where copies of blocks reside are called directory schemes. A directory is a data structure that maintains information on the processors that share a memory block and on its state. The information maintained in the directory could be either centralized or distributed. A Central directory maintains information about all blocks in a central data structure. While Central directory includes everything in one location, it becomes a bottleneck and suffers from large search time. To alleviate this problem, the same information can be handled in a distributed fashion by allowing each memory module to maintain a separate directory. In a distributed directory, the entry associated with a memory block has only one pointer one of the cache that requested the block.

5.1- Protocol Categorization

A directory entry for each block of data should contain a number of pointers to specify the locations of copies of the block. Each entry might also contain a dirty bit to specify whether or not a unique cache has permission to write this memory block. Most directory-based protocols can be categorized under three categories: full-map directories, limited directories, and chained directories.

Full-Map Directories In a full-map setting, each directory entry contains N pointers, where N is the number of processors. Therefore, there could be N cached copies of a particular block shared by all processors. For every memory block, an N-bit vector is maintained, where N equals the number of processors in the shared memory system. Each bit in the vector corresponds to one processor. If the i th bit is set to one, it means that processor i has a copy of this block in its cache. Figure -7 illustrates the fully mapped scheme. In the figure the vector associated with block X in memory indicates that X is in Cache C0 and Cache C2. Clearly the space is not utilized efficiently in this scheme, in particular if not many processors share the same block.

Limited Directories Limited directories have a fixed number of pointers per directory entry regardless of the number of processors. Restricting the number

of simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories. Figure -8 illustrates the limited directory scheme. In this example, the number of copies that can be shared is restricted to two

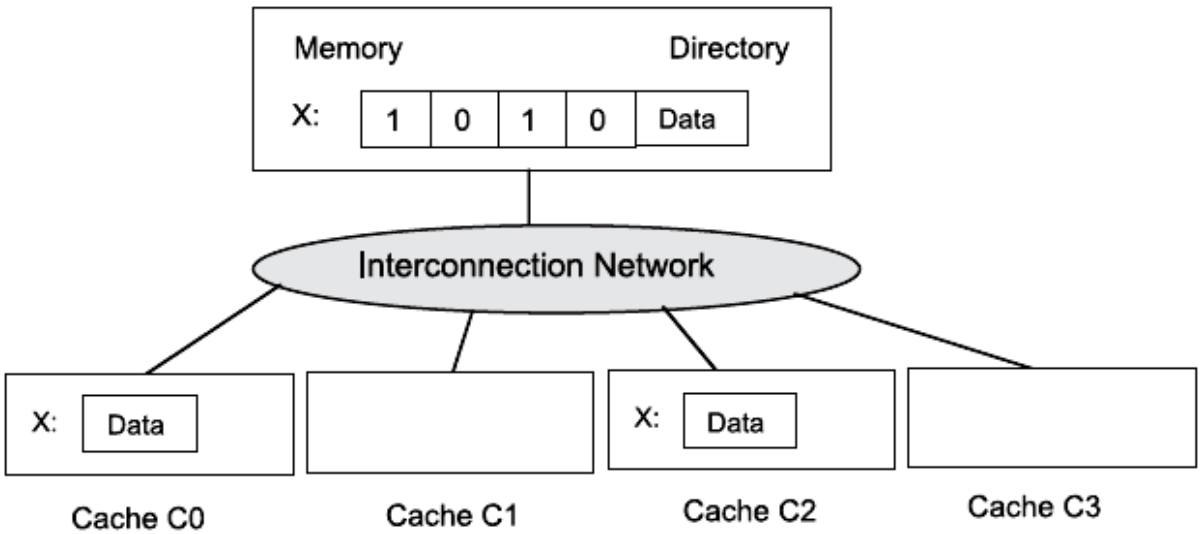

Figure -7 Fully mapped directory.

This is why the vector associated with block X in memory has only two locations. The vector indicates that X is in Cache C0 and Cache C2.

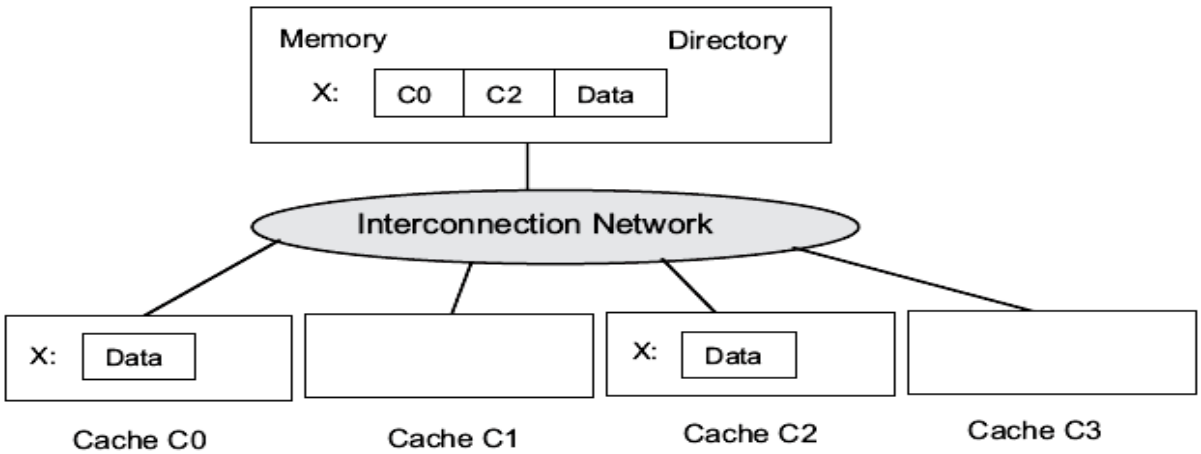

Figure -8 Limited directory (maximum sharing = 2).

Chained Directories Chained directories emulate full-map by distributing the directory among the caches. They are designed to solve the directory size problem without restricting the number of shared block copies. Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers.

Figure -9 shows that the directory entry associated with X has a pointer to Cache C2, which in turn has a pointer to Cache C0. That is, block X exists in the two Caches C0 and Cache C2. The pointer from Cache C0 is pointing to terminator (CT), indicating the end of the list.

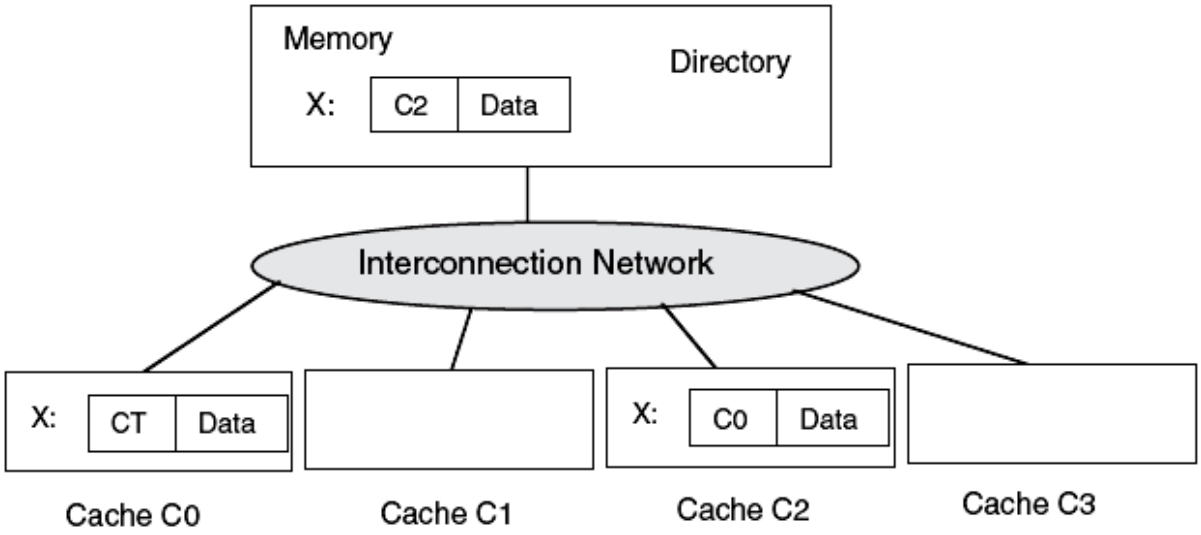

Figure -9 Chained directory.