Lesson: Cache memory Organization

Lesson No. : 04

4.0 Objective

In this lesson we will discuss about bus that is used for interconnections between different processor. We will discuss about use of cache memory in multiprocessor environment and various addressing scheme used for cache memory. The page replacement policy and performance of cache is also measured. Also we will discuss how shared memory concept is used in multiprocessor. Various issues regarding event ordering specially in case of memory events that deal with shared memory creates synchronization problem we will also discuss various models designed to overcome these issues.

4.1 Introduction

In the hierarchy memory cache memory are the fastest memory that lies between registers and RAM . It holds recently used data and/or instructions and has a size varying from few kB to several MB.

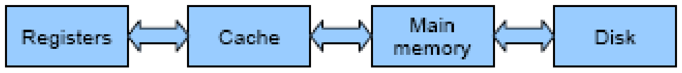


Figure 4.1 Memory structure for a processor

The figure 4.1 shows a cache and main memory structure. A cache consists of C slots and each *slot* in the cache can hold K memory words. Here the main memory with 2^n-1 words i.e., M words with each having a unique n-bit address and cache memory having C*K words where K is the Block size and C are the number of lines. Each word that resides in the cache is a subset of main memory. Since there are more blocks in main memory than number of lines in cache, an individual line cannot be uniquely and permanently dedicated to a particular block. Therefore, each line includes a tag that identifies which particular block of main memory is currently occupying that line of cache. The tag is usually a portion of the main memory address. The cache memory is accessed but by pattern matching on a *tag* stored in the cache.

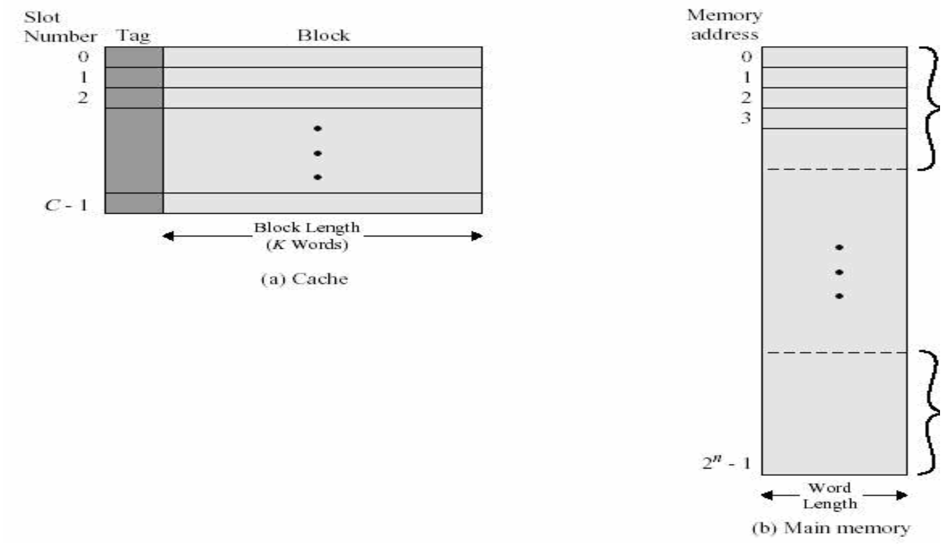


Figure 4.2 Cache / Main memory structure

For the comparison of address generated by CPU the memory controller use some algorithm which determines whether the value currently being addressed in memory is available in the cache. The transformation data from main memory to cache memory is referred as a mapping process. Let us derive an address translation scheme using cache as a linear array of entries, each entry having the following structure as shown in figure 4.3.

A Cache Storage is divided into three fields:

Data - The block of data from memory that is stored in a specific line in the cache

Tag - A small field of length K bits, used for comparison, to check the correct address of data

Valid Bit - A one-bit field that indicates status of data written into the cache.

The N-bit address is produced by the processor to access cache data is divided into three fields:

Tag - A K-bit field that corresponds to the K-bit tag field in each cache entry,

Index - An M-bit field in the middle of the address that points to one cache entry

Byte Offset – L Bits that finds particular data in a line if valid cache is found.

It follows that the length of the virtual address is given by N = K + M + L bits.

Cache Address Translation. As shown in Figure 4.3, we assume that the cache address has length 32 bits. Here, bits 12-31 are occupied by the Tag field, bits 2-11 contain the Index field, and bits 0,1 contain the Offset information. The index points to the line in cache that supposedly contains the data requested by the processor. After the cache line is retrieved, the Tag field in the cache line is compared with the Tag field in the cache address. If the tags do not match, then a cache miss is detected and the comparator outputs a zero value. Otherwise, the comparator outputs a one, which is *and*-ed with the valid bit in the cache row pointed to by the Index field of the cache address. If the valid bit is a one, then the Hit signal output from the *and* gate is a one, and the data in the cached block is sent to the processor. Otherwise a cache miss is registered.

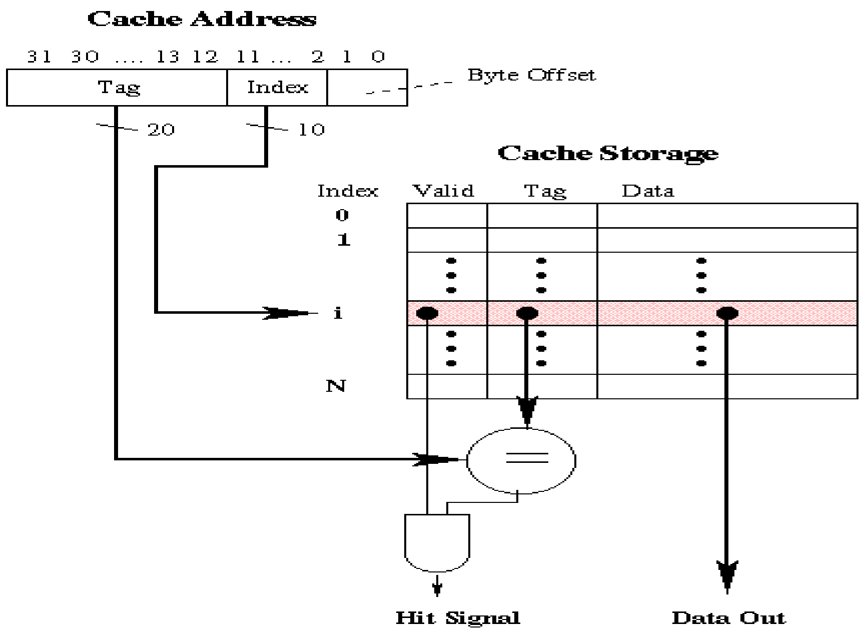


Figure 4.3. Schematic diagram of cache

A cache implements several different policies for retrieving and storing information, one in each of the following categories:

° Fetch policy—determines when information is loaded into the cache.

° Replacement policy—determines what information is purged when space is needed for a new entry.

° Write policy—determines how soon information in the cache is written to lower levels in the memory hierarchy.

4.2 **Cache addressing models**

Most multiprocessor system use private cache associated with different processor.

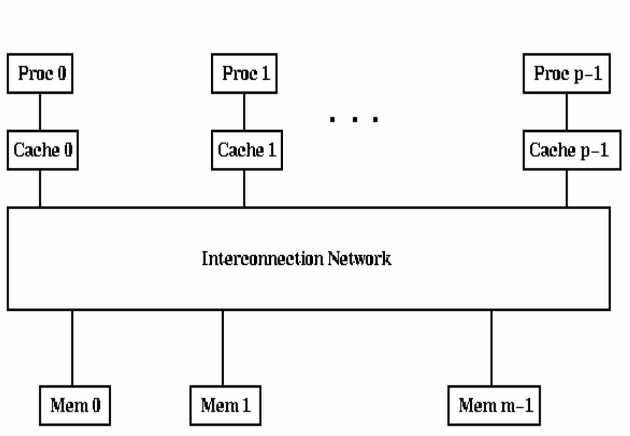


Figure 4.4 A memory hierarchy for a shared memory multiprocessor.

Cache can be addressed either by physical address or virtual address.

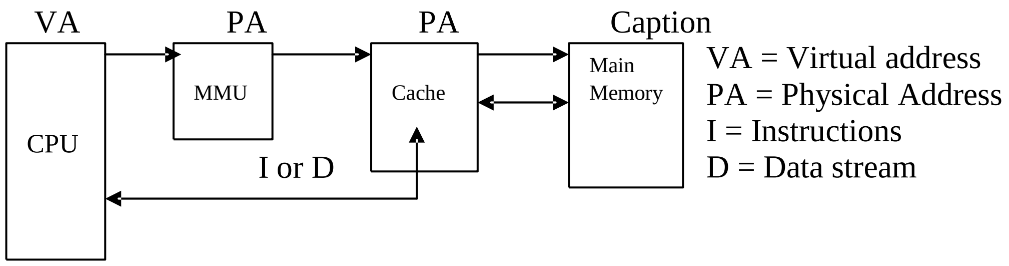
Physical address cache: when cache is addressed by physical address it is called physical address cache. The cache is indexed and tagged with physical address. Cache lookup must occur after address translation in TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion. This provides an advantage that we need no cache flushing, no aliasing problem and fewer cache bugs in OS kernel. The short coming is the slowdown in accessing the cache until the MMU/TLB finishes translating the address.

Advantage of physically addressed caches:

- no cache flushing on a context switch
- no synonym problem (several different virtual addresses can span the same physical addresses : a much better hit ratio between processes)

Disadvantage of physically addressed caches:

- do virtual-to-physical address translation on every access
- increase in hit time because must translate the virtual address before access the cache



4.5 (a) A unified cache accessed by physical address

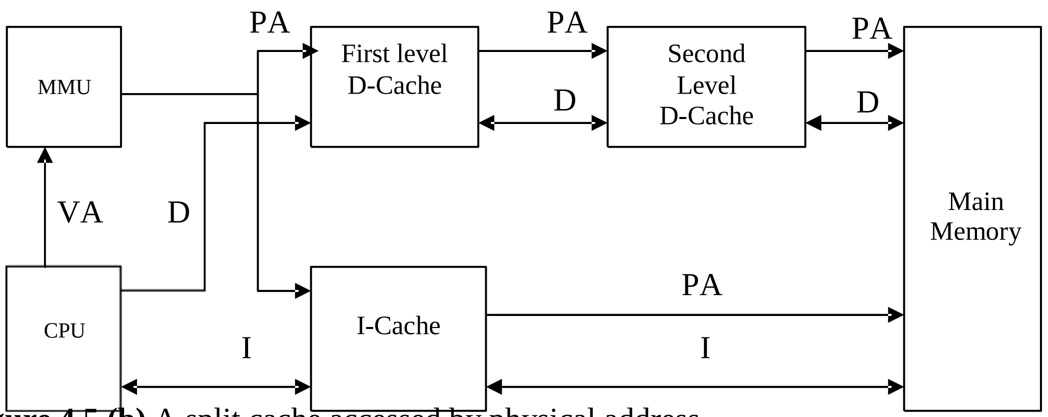


figure 4.5 (b) A split cache accessed by physical address

Virtual Address caches: when a cache is indexed or tagged with virtual address it is called virtual address cache. In this model both cache and MMU translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write back but is not used during the cache lookup operations.

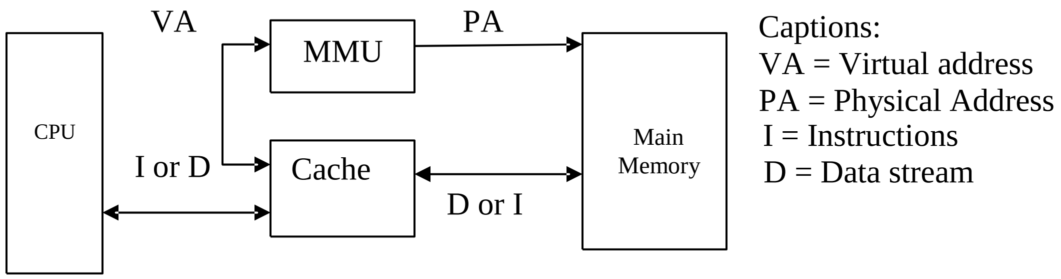
Advantage of virtually-addressed caches

- do address translation only on a cache miss
- faster for hits because no address translation

Disadvantage of virtually-addressed caches

cache flushing on a context switch (example : local data segments will get an erroneous hit for virtual addresses already cached after changing virtual address space, if no cache flushing).

synonym problem (several different virtual addresses cannot span the same physical addresses without being duplicated in cache).



(a) A unified cache accessed by virtual address

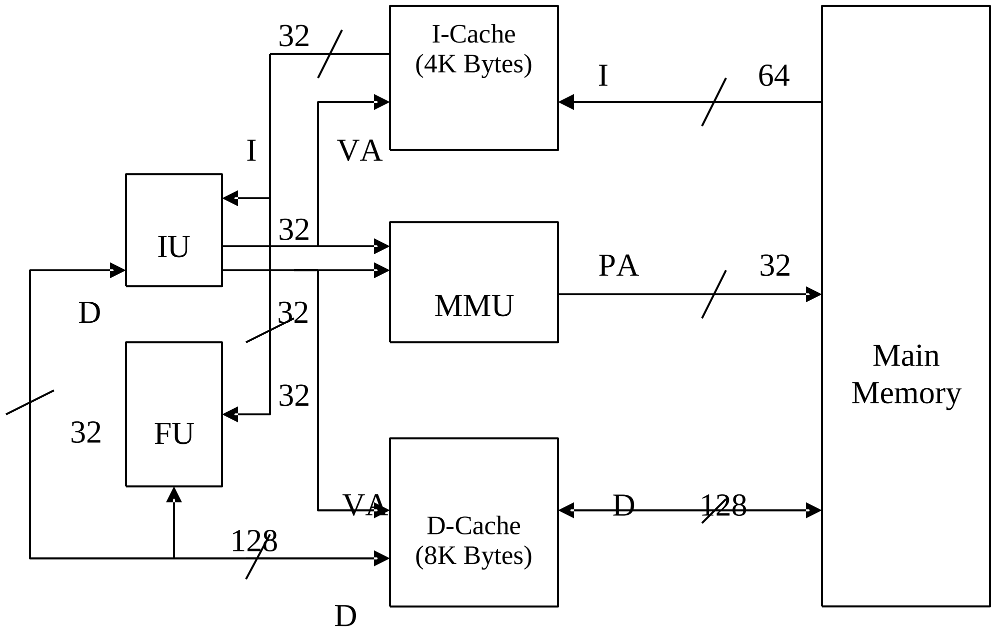


Figure 4.6(b) Virtual address for split cache

Aliasing: The major problem with cache organization in multiprocessor is that multiple virtual addresses can map to a single physical address i.e., different virtual address cache logically addressed data have the same index/tag in the cache. Most processors guarantee that all updates to that single physical address will happen in program order. To deliver on that guarantee, the processor must ensure that only one copy of a physical address resides in the cache at any given time.

4.3 Cache mapping

Caches can be organized according to four different
strategies: ° Direct
° Fully
associative ° Set
associative
° Sectored

4.3.1 Direct-Mapped Caches

The easiest way of organizing a cache memory employs direct mapping that is based on a simple algorithm to map data block i from the main memory into data block j in the cache. There is a one-to-one correspondence between each block of data in the cache and each memory block thus to find a memory block i, then there is one and only one place in the cache where i is stored

If we have 2^n words in main memory and 2^k words in cache memory. In cache memory each word consists of data word and its associated tag. The n-bit memory address is divided into three fields : low order k bits are referred as the index field and used to address a word in the cache. The remaining n-k high-order bits are called the *tag.* The index field is further divided into the *slot* field, which will be used find a particular slot in the cache; and the offset field is used to identify a particular memory word in the slot. When a block is stored in the cache, its *tag* field is stored in the *tag field* of the cache slot.

When CPU generates an address the index field is used to access the cache. The tag field of CPU address is compared with the tag in word read from the cache. If the two tags match, there is a hit and else there is a miss and the required word is read from main memory. Whenever a ``cache miss" occurs, the cache line will be replaced by a new line of information from main memory at an address with the same index but with a different tag.

Lets us understand how direct mapping is implemented with following simple example Figure 4.7. The memory is composed of 32 words and accessed by a 5-bit address. Let the address has a 2-bit tag (set) field, a 2-bit slot (line) field and a 1-bit word field. The cache memory holds 2^2 = 4 lines each having two words. When the processor generates an address, the appropriate line (slot) in the cache is accessed. For example, if the processor generates the 5-bit address 11110_2, line 4 in set 4 is accessed. The memory space is divided into sets and the sets into lines. The Figure 4.7 reveals that there are four possible lines that can occupy cache line 4 lines 4 in set 0, in set 1, in set 2 and set 4. In this example the processor accessed line 4 in set 4. Now "How does the system resolve this issue?"

Figure 4.7 shows how a direct mapped cache resolves the contention between lines. Each line in the cache memory has a tag or label that identifies which set this particular line belongs to. When the processor accesses line 4, the tag belonging to line 4 in the cache is sent to a comparator. At the same time the set field from the processor is also sent to the comparator. If they are the same, the line in the cache is the desired line and a hit occurs. If they are not the same, a miss occurs and the cache must be updated. Figure 4.17 provides a skeleton structure of a direct mapped cache memory system.

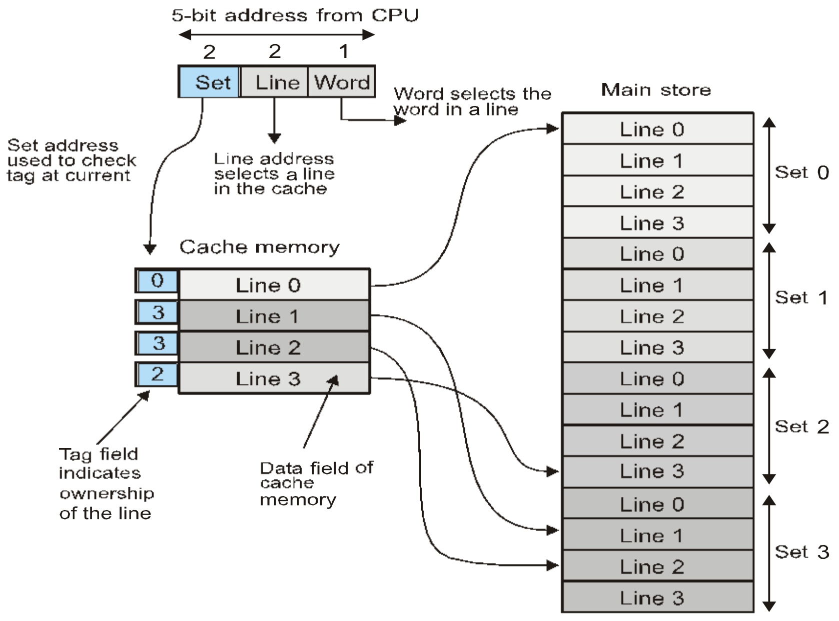

5-bit address from CPU
2
2
1
Set
Line
Word
Word selects the word in a line
Main store
Set address used to check tag at current
Line address selects a line in the cache
Cache memory
0
3
3
2
Line 0
Line 1
Line 2
Line 3
Tag field indicates ownership of the line
Data field of cache memory
Set 0
Set 1
Set 2
Set 3

Figure 4.7 Resolving contention between lines in a direct-mapped cache

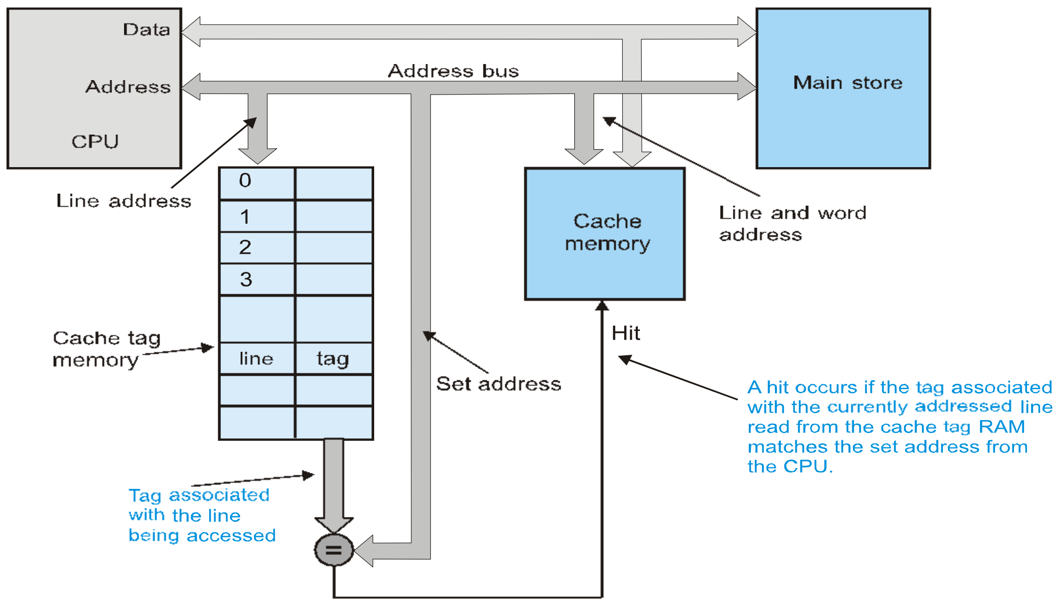

Data
Address
CPU
Address bus
Main store
Line address
0
1
2
3
line
tag
Cache memory
Line and word address
Cache tag memory
Hit
Set address
A hit occurs if the tag associated with the currently addressed line read from the cache tag RAM matches the set address from the CPU.
Tag associated with the line being accessed
=

Figure 4.8 Implementation of direct-mapped cache

The advantage of direct mapping are as follows

It's simplicity.

Both the cache memory and the cache tag RAM are widely available devices.

The direct mapped cache requires no complex line replacement algorithm. If line *x* in set *y* is accessed and a miss takes place, line *x* from set *y* in the main store is loaded into the frame for line *x* in the cache memory and the tag set to *y i.e.*,, there is no decision to be taken regarding which line has to be rejected when a new line is to be loaded.

It inherents parallelism. Since the cache memory holding the data and the cache tag RAM are entirely independent, they can both be accessed simultaneously. Once the tag has been matched and a hit has occurred, the data from the cache will also be valid.

The disadvantage of direct mapping are as follows

it is inflexible

A cache has one restriction *a particular memory address can be mapped into only one cache location also,* all addresses with the same index field are mapped to the same cache location. Consider the following fragment of code:

```
REPEAT
    Get_data
    Compare
UNTIL match OR end_of_data
```

Let the Get data routine and compare routine use two blocks, both these blocks have same index but have different tags are repeated accessed. Consequently, the performance of a direct-mapped cache can be very poor under above circumstances. However, statistical measurements on real programs indicate that the very poor worst-case behavior of direct-mapped caches has no significant impact on their average behavior.

4.3.3 Associative Mapping:

One way of organizing a cache memory which overcomes the limitations of direct mapped cache such that there is no restriction on what data it can contain can be done with associative cache memory. An associative memory is the fastest and most flexible way of cache organization. It stores both the address and the value (data) from main memory in the cache. An associative memory has an *n*-bit input. An address from the processor is divided into three fields: the tag, the line, and the word.The mapping is done with storing tag information in n-bit argument register and comparing it with address tag in each location simultaneously. If the input tag matches a stored tag, the data associated with that location is output. Otherwise the associative memory produces a miss output. Unfortunately, large associative memories are not yet cost-effective. Once the associative cache is full, a new line can be brought in only by overwriting an existing line that requires a suitable line replacement policy. Associative cache memories are efficient because they place no restriction on the data they hold, as permits any location of cache to store any word from main memory.

CPU Address (argument register)

Address	Data
01101001	10010100
10010001	10101010

Figure 4.9 Associative cache

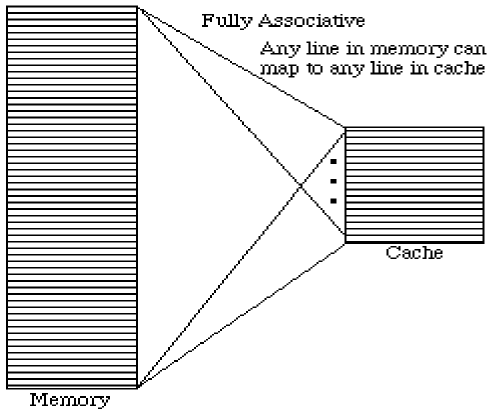
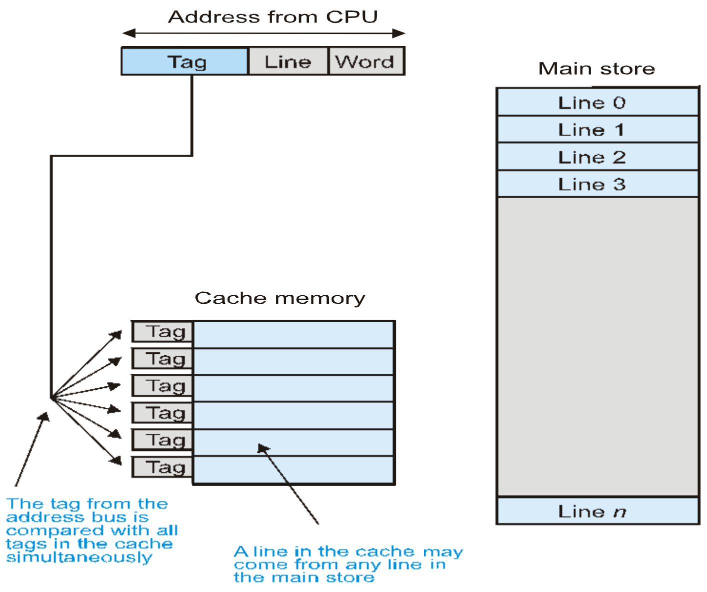


Figure 4.10Associative mapping

All of the comparisons are done simultaneously, so the search is performed very quickly. This type of memory is very expensive, because each memory location must have both a comparator and a storage element. Like the direct mapped cache, the smallest unit of data transferred into and out of the cache is the line. Unlike the direct-mapped cache, there's no relationship between the location of lines in the cache and lines in the main memory.

When the processor generates an address, the word bits select a word location in both the main memory and the cache. The tag resolves which of the lines is actually present. In an associative cache any of the 64K lines in the main store can be located in any of the lines in the cache. Consequently, the associative cache requires a 16-bit tag to identify one of the 2^{16} lines from the main memory. Because the cache's lines are not ordered, the tags are not ordered, it may be anywhere in the cache or it may not be in the cache.



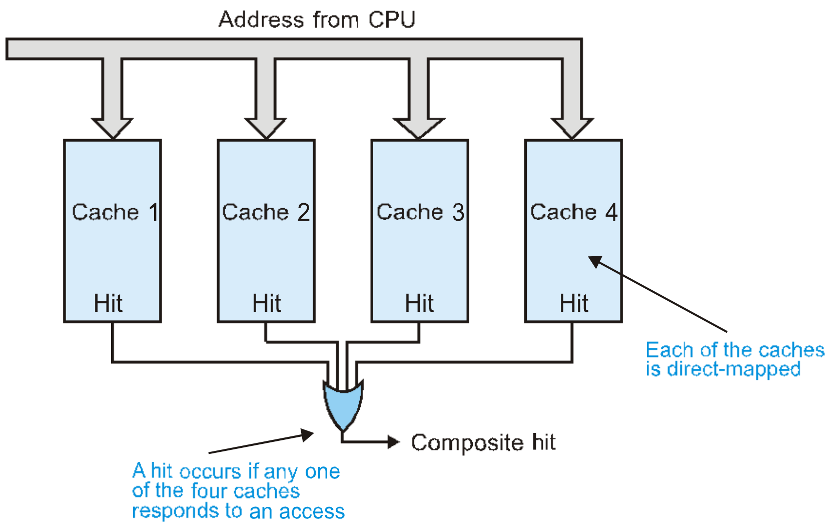
Address from CPU
Tag
Line
Word
Main store
Line 0
Line 1
Line 2
Line 3
Line n
Cache memory
Tag
Tag
Tag
Tag
Tag
Tag
The tag from the address bus is compared with all tags in the cache simultaneously
A line in the cache may come from any line in the main store

Figure 4.11 Associative-mapped cache

4.3.4 Set associative Mapping:

Most computers use set associative mapping technique as it is a compromise between the direct-mapped cache and the fully associative cache. In a set associative cache memory several direct-mapped caches connected in parallel. Let to find memory block b in the cache, there are n entries in the cache that can contain b we say that this type of cache is called n-*way set associative*. For example, if n = 2, then we have a two-way set associative cache. This is the simplest arrangement and consists of two direct-mapped cache memories. Thus for n parallel sets, a n-way comparison is performed in parallel against all members of the set. Usually $n = 2^k$, for *k* = 1, 2, 4 are chosen for a set associative cache (*k* = 0 corresponds to direct mapping). As n is small (typically 2 to 14), the logic required to perform the comparison is not complex. This is a widely used technique in practice (e.g. 80486 uses 4-way, P4 uses 2-way for the instruction cache, 4-way for the data cache).

Figure 4.22 describes the common 4-way set associative cache. When the processor accesses memory, the appropriate line in each of four direct-mapped caches is accessed simultaneously. Since there are four lines, a simple associative match can be used to determine which (if any) of the lines in cache are to supply the data. In figure 4.22 the hit output from each direct-mapped cache is fed to an OR gate which generates a hit if any of the caches generate a hit.



Address from CPU
Cache 1
Cache 2
Cache 3
Cache 4
Hit
Hit
Hit
Hit
Each of the caches is direct-mapped
Composite hit
A hit occurs if any one of the four caches responds to an access

Figure 4.12 Set associative-mapped cache

4.3.4 Sector mapped cache memory

The idea is to partition both the cache and memory into fixed size sectors. Thus in a sectored cache, main memory is partitioned into sectors, each containing several blocks. The cache is partitioned into sector frames, each containing several lines. (The number of lines/sector frame = the number of blocks/sector.) As shown in figure below sector size is of 16 block. Each sector can be mapped to any of the sector frame with full associative at the sector level.

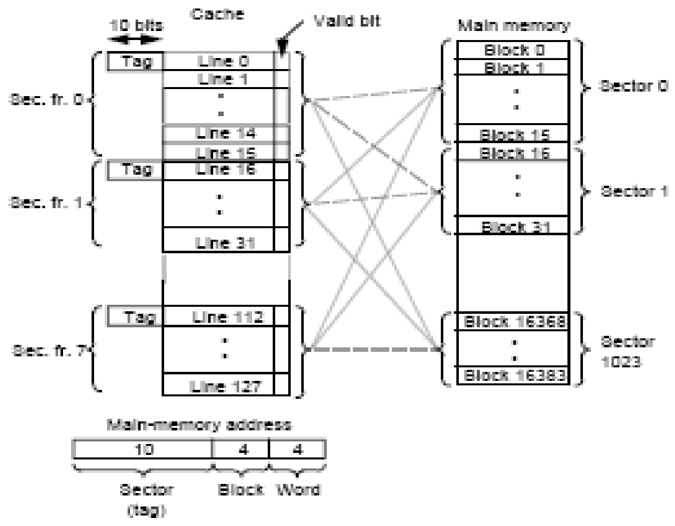


Figure 4.13 Sector mapped memory

Each sector can be placed in any of the available sector frame. The memory requests are destined for blocks not for sectors. This can be filtered out by comparing the sector tag in the memory address with all sector tags using fully associative search.

When block b of a new sector c is brought in,

• it is brought into line b within some sector frame f, and

• the rest of the lines in sector frame f are marked invalid.

Thus, if there are S sector frames, there are S choices of where to place a block.

4.3.5 CACHE performance Issues

As far as the performance of cache is considered the trade off exist among the cache size, set number, block size and memory speed. Important aspect in cache designing with regard to performance are :

a. **the cycle count** : This refers to the number of basic machine cycles needed for cache access, update and coherence control. This count is affected by underlying static or dynamic RAM technology, the cache organization and the cache hit ratios. The write through or write back policy also affect the cycle count. The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of above cache parameters.

b. **Hit ratio:** The processor generates the address of a word to be read and send it to cache controller, if the word is in the cache it generates a Hit signal and also deliver it to the processor. If the data is not found in the cache, then it generates a MISS signal and that data is delivered to the processor from main memory, and simultaneously loaded into the cache. The hit ratio is number of hits divided by total number of CPU references to memory (hits plus misses). When cache size approaches

c. **Effect of Block Size:** With a fixed cache size, cache performance is sensitive to the block size. This block size is determined mainly by the temporal locality in typical program.

d. Effect of set number in set associative number.

4.4 Cache replacement algorithm

When a new block is brought into cache, one of the existing blocks must be replaced. The obvious question arise is which page to be replaced? With direct mapping, the solution is easy as we have not choice. But in other circumstances, we do. The three most commonly used algorithms are *Least Recently Used*, *First in First out* and *Random*.

Random -- The optimal algorithm is called *random replacement*, whereby a location to which a block is to be written in cache is chosen at random from the range of cache indices. The random replacement strategy usually implemented using a random number generator. In a 2-way set associative cache, this can be accomplished with a single modulo 2 random variable obtained, from an internal clock

First in, first out (FIFO) -- here the first value *stored* in the cache is the index position representing value to be replaced. For a 2-way set associative cache, this replacement strategy can be implemented by setting a pointer to the previously loaded word each time a new word is *stored* in the cache; this pointer need only be a single bit.

Least recently used (LRU) -- here the value which was actually used least recently is replaced. In general, it is more likely that the most recently used value will be the one required in the near future. This approach, while not always optimal, is intuitively attractive from the perspective of temporal locality. That is, a given program will likely not access a page or block that has not been accessed for some time. The LRU replacement algorithm requires that each cache or page table entry have a *timestamp*. This is a combination of date and time that uniquely identifies the entry as having been written at a particular time. Given a timestamp t with each of N entries, LRU merely finds the minimum of the cached timestamps, as

$t_{min} = min\{t_i : i = 1..N\}$.

The cache or page table entry having $t = t_{min}$ is then overwritten with the new entry.

For a 2-way set associative cache, this is readily implemented by setting a special bit called the ``USED" bit for the other word when a value is *accessed* while the corresponding bit for the word which was accessed is reset. The value to be replaced is then the value with the USED bit set. This replacement strategy can be implemented by adding a single USED bit to each cache location. The LRU strategy operates by setting a bit in the other word when a value is *stored* and resetting the corresponding bit for the new word. For an *n*-way set associative cache, this strategy can be implemented by storing a modulo *n* counter with each data word.

4.5 Cache Coherence and Synchronization

4.5.1Cache coherence problem

An important problem that must be addressed in many parallel systems - any system that allows multiple processors to access (potentially) multiple copies of data - is *cache coherence*. The existence of multiple cached copies of data creates the possibility of inconsistency between a cached copy and the shared memory or between cached copies themselves.

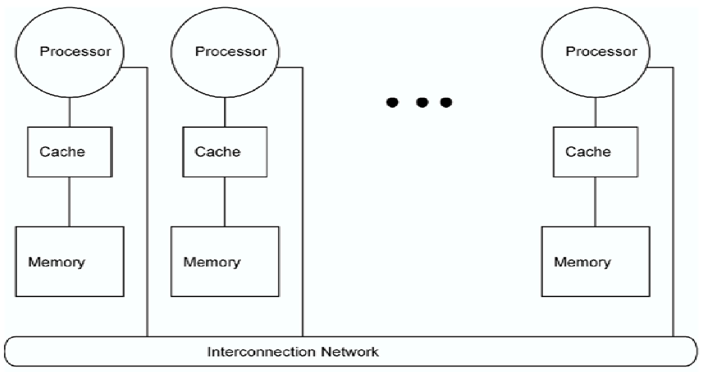


Figure 4.14 cache coherence problem in multiprocessor

There are three common sources of cache inconsistency:

- Inconsistency in data sharing : In a memory hierarchy for a multiprocessor system data inconsistency may occur between adjacent levels or within the same level. The cache inconsistency problem occurs only when multiple private cache are used. Thus it is, the possible that a wrong data being accessed by one processor because another processor has changed it, and not all changes have yet been propagated. Suppose we have two processors, A and B, each of which is dealing with memory word X, and each of which has a cache. If processor A changes X, then the value seen by processor B in *its own cache* will be wrong, *even if processor A also changes the value of X in main memory* (which it - ultimately - should).

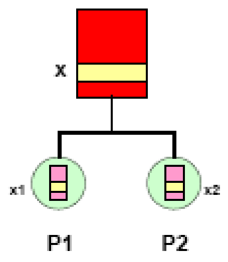


Figure 4.15 Cache coherence problem

In above example initially, x1 = x2 = X = 5.

P1 writes X:=10 using *write-through*.

P2 now reads X and uses its local copy x2, but finds that X is still 5.

Thus P2 does not know that P1 modified X.

Thus the cache inconsistency problem occurs when multiple private cache are used and especially the problem arose by writing the shared variables.

- Process migration(even if jobs are independent): This problem occurs when a process containing shared variable X migrates from process 1 to process2 using the write back cache on the right. Thus another important aspect of coherence is *serialization* of writes - that is, if two processors try to write 'simultaneously', then (i) the writes happen sequentially (and it doesn't really matter who gets to write first - provided we have sensible arbitration); and (ii) *all processors see the writes as occurring in the same order*. That is, if processors A and B both write to X, with A writing first, then any other processors (C, D, E) *all* see the *same* thing.

- DMA I/O – this inconsistency problem occur during the I/O operation that bypass the cache. This problem is present even in a uniprocessor and can be removed by OS cache flushes)

In practice, these issues are managed by a memory bus, which by its very nature ensures write serialization, and also allows us to broadcast invalidation signals (we essentially just put the memory address to be invalidated on the bus). We can add an extra *valid* bit to cache tags to mark then invalid. Typically, we would use a write-back cache, because it has much lower memory bandwidth requirements. Each processor must keep track of which cache blocks are *dirty* - that is, that it has written to - again by adding a bit to the cache tag. If it sees a memory access for a word in a cache block it has marked as dirty, it intervenes and provides the (updated) value. There are numerous other issues to address when considering cache coherence.

One approach to maintaining coherence is to recognize that not every location needs to be shared (and in fact most don't), and simply reserve some space for non-cacheable data such as semaphores, called a coherency domain.

Using a fixed area of memory, however, is very restrictive. Restrictions can be reduced by allowing the MMU to tag segments or pages as non-cacheable. However, that requires the OS, compiler, and programmer to be involved in specifying data that is to be coherently shared. For example, it would be necessary to distinguish between the sharing of semaphores and simple data so that the data can be cached once a processor owns its semaphore, but the semaphore itself should never be cached.

In order to remove this data inconsistency there are a number of approaches based on hardware and software techniques few are given below:

- No caches is used which is not a feasible solution
- Make shared-data non-cacheable this is the simplest software solution but produce low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections, this is relatively simple technique but has low performance if synchronization is not frequent
- hardware *cache coherence this can be achieved by making* memory and caches coherent (consistent) with each other, in other words if the memory and other processors see writes then without intervention of the to software

- absolute coherence all copies of each block have same data at all times
- It is not necessary what is required is *appearance of absolute coherence that is done by making* temporary incoherence is OK (e.g., write-back cache)

In general a cache coherence protocols consist of the set of possible states in local caches, the state in shared memory and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. There are basically two kinds of protocols depends on how writes is handled

4.5.2 Snooping Cache Protocol (for bus-based machines);

With a bus interconnection, cache coherence is usually maintained by adopting a "snoopy protocol", where each cache controller "snoops" on the transactions of the other caches and guarantees the validity of the cached data. In a (single-) multi-stage network, however, the unavailability of a system "bus" where transactions are broadcast makes snoopy protocols not useful. Directory based schemes are used in this case.

In case of snooping protocol processors perform some form of *snooping* - that is, keeping track of other processor's memory writes. ALL caches/memories see and react to ALL bus events. The protocol relies on global visibility of requests (ordered broadcast). This allows the processor to make state transitions for its cache-blocks.

Write Invalidate protocol

The states of a cache block copy changes with respect to read, write and replacement operations in the cache. The most common variant of snooping is a *write invalidate protocol*. In the example above, when processor A writes to X, it broadcasts the fact and all other processors with a copy of X in their cache mark it invalid. When another processor (B, say) tries to access X again then there will be a cache miss and either

(i) in the case of a write-through cache the value of X will have been updated (actually, it might not because not enough time may have elapsed for the memory write to complete - but that's another issue); or

(ii) in the case of a write-back cache processor A must spot the read request, and substitute the *correct* value for X.

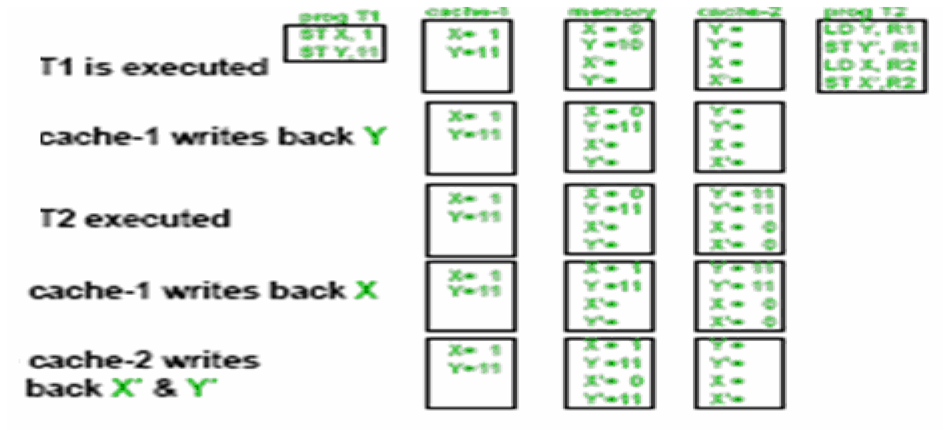


Figure 6.16 Write back with cache

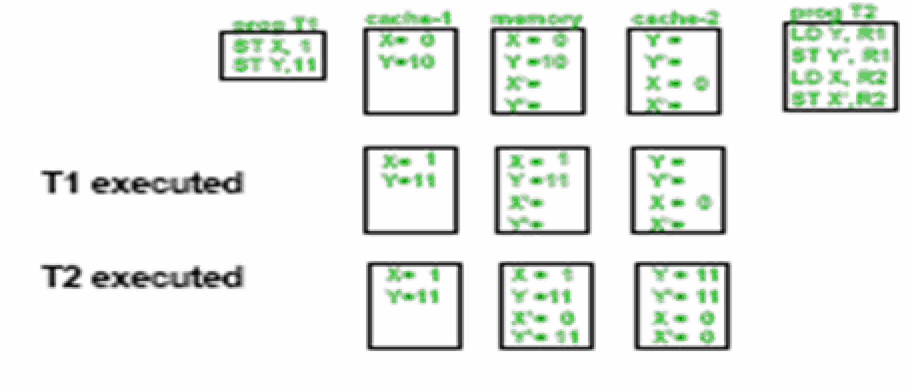
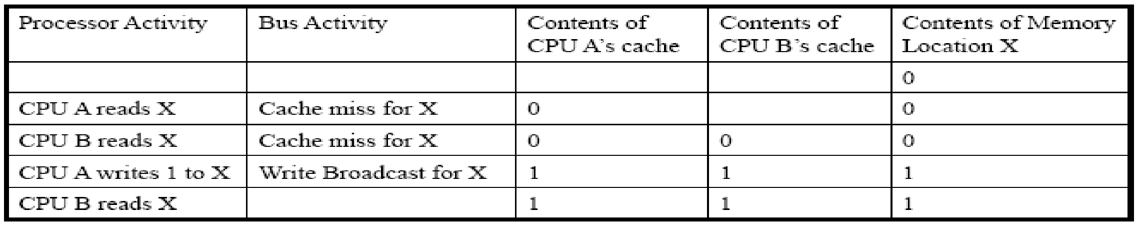


Figure 6. 17 Write through with cache

An alternative (but less-common) approach is *write broadcast*. This is intuitively a little more obvious - when a cached value is changed, the processor that changed it broadcasts the new value to all other processors. They then update their own cached values. The trouble with this scheme is that it uses up more memory bandwidth. A way to cut this is to observe that many memory words are *not* shared - that is, they will only appear in one cache. If we keep track of which words are shared and which are not, we can reduce the amount of broadcasting necessary. There are two main reasons why more memory bandwidth is used: in an invalidation scheme, only the *first* change to a word requires an invalidation signal to be broadcast, whereas in a write broadcast scheme all changes must be signaled; and in an invalidation scheme only the *first* change to *any* word in a cache block must be signaled, whereas in a write broadcast scheme every word that is written must be signaled. On the other hand, in a write broadcast scheme we do not end up with a cache miss when trying to access a changed word, because the cached copy will have been updated to the correct value.



Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

Figure 6.18 write back with broadcast

If different processors operate on different data items, these can be cached.

1. Once these items are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.

2. If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local.

In both cases, the coherence protocol does not add any overhead.

4.5.3Write-through vs. Write-back

In a write-back cache, the snooping logic must also watch for reads that access main memory locations corresponding to dirty locations in the cache (locations that have been changed by the processor but not yet written back).

At first it would seem that the simplest way to maintain coherence is to use a write-through policy so that every cache can snoop every write. However, the number of extra writes can easily saturate a bus. The solution to this problem is to use a write-back policy, but that leads to additional problems because there can be multiple writes that do not go to the bus, leading to incoherent data.

One approach is called write-once. In this scheme, the first write is a write-through to signal invalidation to other caches. After that, further writes can occur in write-back mode as long as there is no invalidation. Essentially, the first write takes ownership of the data, and another write from another processor must first deal with the invalidation and may then take ownership. Thus, a cache line has four states:

- Invalid
- Valid unwritten (valid)
- Valid written once (reserved)
- Valid written multiple (dirty)

The last two states indicate ownership. The trouble with this scheme is that if a non-owner frequently accesses an owned shared value, it can slow down to main memory speed or slower, and generate excessive bus traffic because all accesses must be to the owning cache, and the owning cache would have to perform a broadcast on its next write to signal that the line is again invalid.

One solution is to grant ownership to the first processor to write to the location and not allow reading directly from the cache. This eliminates the extra read cycles, but then the cache must write-through all cycles in order to update the copies.

We can change the scheme so that when a write is broadcast, if any other processor has a snoop hit, it signals this back to the owner. Then the owner knows it must write through again. However, if no other processor has a copy (signals snooping), it can proceed to write privately. The processor's cache must then snoop for read accesses from other processors and respond to these with the current data, and by marking the line as snooped. The line can return to private status once a write-through results in a no-snoop response.

One interesting side effect of ownership protocols is that they can sometimes result in a speedup greater than the number of processors because the data resides in faster memory. Thus, other processors gain some speed advantage on misses because instead of fetching from the slower main memory, they get data from another processor's fast cache. However, it takes a fairly unusual pattern of access for this to actually be observed in real system performance.

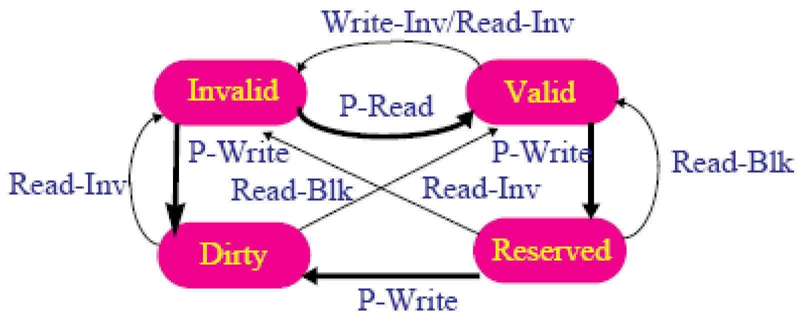


Figure 6.19 write once protocol

Disadvantages:

- If multiple processors read and update the same data item, they generate coherence functions across processors.

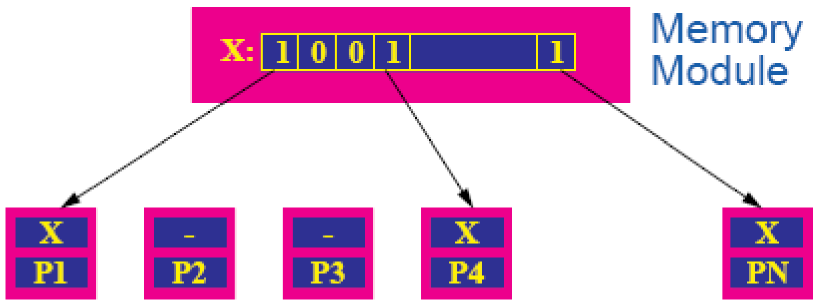
 Since a shared bus has a finite bandwidth, only a constant

Rather than flush the cache completely, hardware can be provided to "snoop" on the bus, watching for writes to main memory locations that are cached.

Another approach is to have the DMA go through the cache, as if the processor is writing it to memory. This results in all valid cache locations. However, any processor cache accesses are stalled during that time, and it clearly does not work well in a multiprocessor, as it would require copies being written to all caches and a protocol for write-back to memory that avoids inconsistency.

4.5.4 Directory-based Protocols

When a multistage network is used to build a large multiprocessor system, the snoopy cache protocols must be modified. Since broadcasting is very expensive in a multistage network, consistency commands are sent only to caches that keep a copy of the block. This leads to *Directory Based protocols*. A directory is maintained that keeps track of the sharing set of each memory block. Thus each bank of main memory can keep a directory of all caches that have copied a particular line (block). When a processor writes to a location in the block, individual messages are sent to any other caches that have copies. Thus the Directory-based protocols selectively send invalidation/update requests to only those caches having copies—the sharing set leading the network traffic limited only to essential updates. Proposed schemes differ in the latency with which memory operations are performed and the implementation cost of maintaining the directory. The memory must keep a bit-vector for each line that has one bit per processor, plus a bit to indicate ownership (in which case there is only one bit set in the processor vector).



.figure 6.20 Directory based protocol

These bitmap entries are sometimes referred to as the presence bits. Only processors that hold a particular block (or are reading it) participate in the state transitions due to coherence operations. Note that there may be other state transitions triggered by processor read, write, or flush (retiring a line from cache) but these transitions can be handled locally with the operation reflected in the presence bits and state in the directory. If different processors operate on distinct data blocks, these blocks become dirty in the respective caches and all operations after the first one can be performed locally.

If multiple processors read (but do not update) a single data block, the data block gets replicated in the caches in the shared state and subsequent reads can happen without triggering any coherence overheads.

Various directory-based protocols differ mainly in how the directory maintains information and what information is stored. Generally speaking the directory may be central or distributed. Contention and long search times are two drawbacks in using a central directory scheme. In a distributed-directory scheme, the information about memory blocks is distributed. Each processor in the system can easily "find out" where to go for "directory information" for a particular memory block. Directory-based protocols fall under one of three categories:

Full-map directories, limited directories, and chained directories.

This full-map protocol is extremely expensive in terms of memory as it store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data.. It thus defeats the purpose of leaving a bus-based architecture.

A limited-map protocol stores a small number of processor ID tags with each line in main memory. The assumption here is that only a few processors share data at one time. If there is a need for more processors to share the data than there are slots provided in the directory, then broadcast is used instead.

Chained directories have the main memory store a pointer to a linked list that is itself stored in the caches. Thus, an access that invalidates other copies goes to memory and then traces a chain of pointers from cache to cache, invalidating along the chain. The actual write operation stalls until the chain has been traversed. Obviously this is a slow process.

Duplicate directories can be expensive to implement, and there is a problem with keeping them consistent when processor and bus accesses are asynchronous. For a write-through cache, consistency is not a problem because the cache has to go out to the bus anyway, precluding any other master from colliding with its access.

But in a write-back cache, care must be taken to stall processor cache writes that change the directory while other masters have access to the main memory.

On the other hand, if the system includes a secondary cache that is inclusive of the primary cache, a copy of the directory already exists. Thus, the snooping logic can use the secondary cache directory to compare with the main memory access, without stalling the processor in the main cache. If a match is found, then the comparison must be passed up to the primary cache, but the number of such stalls is greatly reduced due to the filtering action of the secondary cache comparison.

A variation on this approach that is used with write-back caches is called dirty inclusion, and simply requires that when a primary cache line first becomes dirty, the secondary line is similarly marked. This saves writing through the data, and writing status bits on every write cycle, but still enables the secondary cache to be used by the snooping logic to monitor the main memory accesses. This is especially important for a read-miss, which must be passed to the primary cache to be satisfied.

The previous schemes have all relied heavily on broadcast operations, which are easy to implement on a bus. However, buses are limited in their capacity and thus other structures are required to support sharing for more than a few processors. These structures may support broadcast, but even so, broadcast-based protocols are limited.

The problem is that broadcast is an inherently limited means of communication. It implies a resource that all processors have access to, which means that either they contend to transmit, or they saturate on reception, or they have a factor of N hardware for dealing with the N potential broadcasts.

Snoopy cache protocols are not appropriate for large-scale systems because of the bandwidth consumed by the broadcast operations

In a multistage network, cache coherence is supported by using cache directories to store information on where copies of cache reside.

A cache coherence protocol that does not use broadcast must store the locations of all cached copies of each block of shared data. This list of cached locations whether centralized or distributed is called a cache directory. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block.

Distributed directory schemes

In scalable architectures, memory is physically distributed across processors. The corresponding presence bits of the blocks are also distributed. Each processor is responsible for maintaining the coherence of its own memory blocks. Since each memory block has an owner its directory location is implicitly known to all processors. When a processor attempts to read a block for the first time, it requests the owner for the block. The owner suitably directs this request based on presence and state information locally available. When a processor writes into a memory block, it propagates an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block. Note that the communication overhead associated with state update messages is not reduced. Distributed directories permit O(p) simultaneous coherence operations, provided the underlying network can sustain the associated state update messages. From this point of view, distributed directories are inherently more scalable than snoopy systems or centralized directory systems. The latency and bandwidth of the network become fundamental performance bottlenecks for such systems.

4.6 Keywords

cache A high-speed memory, local to a single processor , whose data transfers are carried out automatically in hardware. Items are brought into a cache when they are referenced, while any changes to values in a cache are automatically written when they are no longer needed, when the cache becomes full, or when some other process attempts to access them. Also To bring something into a cache.

cache consistency The problem of ensuring that the values associated with a particular variable in the *caches* of several processors are never visibly different.

associative memory: Memory that can be accessed by content rather than by address; content addressable is often used synonymously. An associative memory permits its user to specify part of a pattern or key and retrieve the values associated with that pattern.

direct mapping :A cache that has a set associativity of one so that each item has a unique place in the cache at which it can be stored.

4.7 Summary

In this lesson we had learned how cache memory in multiprocessor is organized and how its address are generated both for physical and virtual address. Various techniques of cache mapping are discussed.

Mapping technique	Advantage	disadvantage
Direct Mapping	Fast lookup (only one comparison needed). Cheap hardware (no associative comparison). Easy to decide	Contention for lines
Fully associative	Minimal contention for lines. Wide variety of replacement algorithms feasible.	The most expensive of all organizations, due to the high cost of associative-comparison hardware.

Set associative mapping trade off advantage and disadvantage of direct and fully associative mapping.

We had discussed about the shared memory organization and how consistency is maintained in it. There are various issues of synchronization and event handling on which various consistency models are designed. Various techniques through which cache coherence is maintained are discussed. Bus based systems are not scalable and not efficient for the processor to snoop and handle the traffic. Directories based system is used in cache coherence for large MPs *Cache coherency protocols maintain exclusive writes in a multiprocessor. Memory consistency policies determine how different processors observe the ordering of reads and writes to memory.* Snoopy caches are typically associated with multiprocessor systems based on broadcast interconnection networks such as a bus or a ring. All processors snoop on (monitor) the bus for transactions. Directory based systems the global memory is augmented with a directory that maintains a bitmap representing cache-blocks and the processors at which they are cached.

4.8 Self assessment questions

1. With diagram, explain the interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory and shared peripherals.
2. Discuss advantage and disadvantage of various cache mapping techniques
3. Discuss different page replacement polices.
4. Describe the Cache coherence problems in data sharing and in process migration.
5. Draw and explain 2 state-transition graphs for a cache block using write-invalidate snoopy protocols.
6. Explain the Goodman's write-once cache coherence protocol using the write-invalidate policy on write-back caches.
7. Discuss the basic concept of a directory-based cache coherence scheme.
8. Mention and explain the three types of cache directory protocols.

4.9 **References/Suggested readings**

Advance Computer architecture: Kai Hwang