# Cache coherency

In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands (data) are propagated throughout the system in a timely fashion.[1]

The following are the requirements for cache coherence:[2]

**Write Propagation**

Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches.

**Transaction Serialization**

Reads/Writes to a single memory location must be seen by all processors in the same order.

Theoretically, coherence can be performed at the load/store granularity. However, in practice it is generally performed at the granularity of cache blocks.

Definition

Coherence defines the behavior of reads and writes to a single address location.

One type of data occurring simultaneously in different cache memory is called cache coherence, or in some systems, global memory.

In a multiprocessor system, consider that more than one processor has cached a copy of the memory location X. The following conditions are necessary to achieve cache coherence:

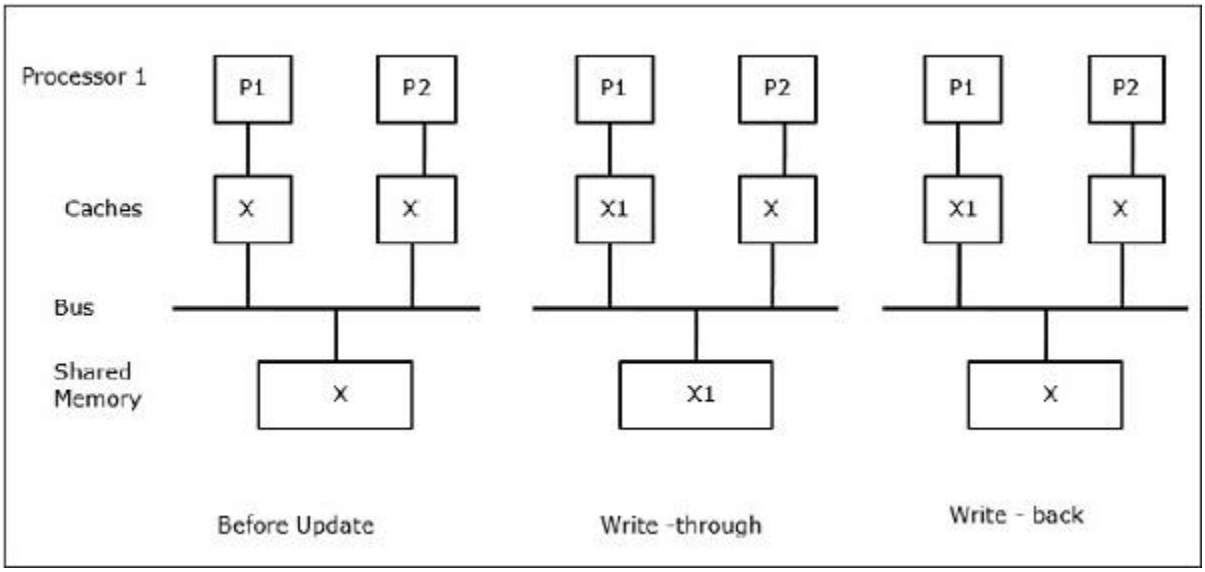
1. In a read made by a processor P to a location X that follows a write by the same processor P to X, with no writes to X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P.
2. In a read made by a processor P1 to location X that follows a write by another processor P2 to X, with no other writes to X made by any processor occurring between the two accesses and with the read and write being sufficiently separated, X must always return the value written by P2. This condition defines the concept of coherent view of

memory. Propagating the writes to the shared memory location ensures that all the caches have a coherent view of the memory. If processor P1 reads the old value of X, even after the write by P2, we can say that the memory is incoherent.

The above conditions satisfy the Write Propagation criteria required for cache coherence. However, they are not sufficient as they do not satisfy the Transaction Serialization condition. To illustrate this better, consider the following example:

A multi-processor system consists of four processors - P1, P2, P3 and P4, all containing cached copies of a shared variable *S* whose initial value is 0. Processor P1 changes the value of *S* (in its cached copy) to 10 following which processor P2 changes the value of *S* in its own cached copy to 20. If we ensure only write propagation, then P3 and P4 will certainly see the changes made to *S* by P1 and P2. However, P3 may see the change made by P1 after seeing the change made by P2 and hence return 10 on a read to *S*. P4 on the other hand may see changes made by P1 and P2 in the order in which they are made and hence return 20 on a read to *S*. The processors P3 and P4 now have an incoherent view of the memory.

As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates **cache coherence problem**. **Cache coherence schemes** help to avoid this problem by maintaining a uniform state for each cached block of data.




Let X be an element of shared data which has been referenced by two processors, P1 and P2. In the beginning, three copies of X are consistent. If the processor P1 writes a new data X1 into the cache, by using **write-through policy**, the same

copy will be written immediately into the shared memory. In this case, inconsistency occurs between cache memory and the main memory. When a **write-back policy** is used, the main memory will be updated when the modified data in the cache is replaced or invalidated.

In general, there are three sources of inconsistency problem −

- Sharing of writable data
- Process migration
- I/O activity

Therefore, in order to satisfy Transaction Serialization, and hence achieve Cache Coherence, the following condition along with the previous two mentioned in this section must be met:

- Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, from any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.

The alternative definition of a coherent system is via the definition of sequential consistency memory model: "the cache coherent system must appear to execute all threads' loads and stores to a *single* memory location in a total order that respects the program order of each thread". Thus, the only difference between the cache coherent system and sequentially consistent system is in the number of address locations the definition talks about (single memory location for a cache coherent system, and all memory locations for a sequentially consistent system).

Another definition is: "a multiprocessor is cache consistent if all writes to the same memory location are performed in some sequential order".

Rarely, but especially in algorithms, coherence can instead refer to the locality of reference. Multiple copies of same data can exist in different cache simultaneously and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

## Coherence mechanisms

The two most common mechanisms of ensuring coherency are *snooping* and *directory-based*, each having their own benefits and

drawbacks. Snooping based protocols tend to be faster, if enough bandwidth is available, since all transactions are a request/response seen by all processors. The drawback is that snooping isn't scalable. Every request must be broadcast to all nodes in a system, meaning that as the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides must grow. Directories, on the other hand, tend to have longer latencies (with a 3 hop request/forward/respond) but use much less bandwidth since messages are point to point and not broadcast. For this reason, many of the larger systems (>64 processors) use this type of cache coherence.

**Snooping**

*Main article: Bus snooping*

First introduced in 1983, snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached. The *write-invalidate protocols* and *write-update protocols* make use of this mechanism.

For the snooping mechanism, a snoop filter reduces the snooping traffic by maintaining a plurality of entries, each representing a cache line that may be owned by one or more nodes. When replacement of one of the entries is required, the snoop filter selects for the replacement the entry representing the cache line or lines owned by the fewest nodes, as determined from a presence vector in each of the entries. A temporal or other type of algorithm is used to refine the selection if more than one cache line is owned by the fewest nodes.

**Directory-based**

*Main article: Directory-based cache coherence*

In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory acts as a filter through which the processor must ask permission to load an entry

from the primary memory to its cache. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

Distributed shared memory systems mimic these mechanisms in an attempt to maintain consistency between blocks of memory in loosely coupled systems.

## Hardware Synchronization Mechanisms

Synchronization is a special form of communication where instead of data control, information is exchanged between communicating processes residing in the same or different processors.

Multiprocessor systems use hardware mechanisms to implement low-level synchronization operations. Most multiprocessors have hardware mechanisms to impose atomic operations such as memory read, write or read-modify-write operations to implement some synchronization primitives. Other than atomic memory operations, some inter-processor interrupts are also used for synchronization purposes.

## Cache Coherency in Shared Memory Machines

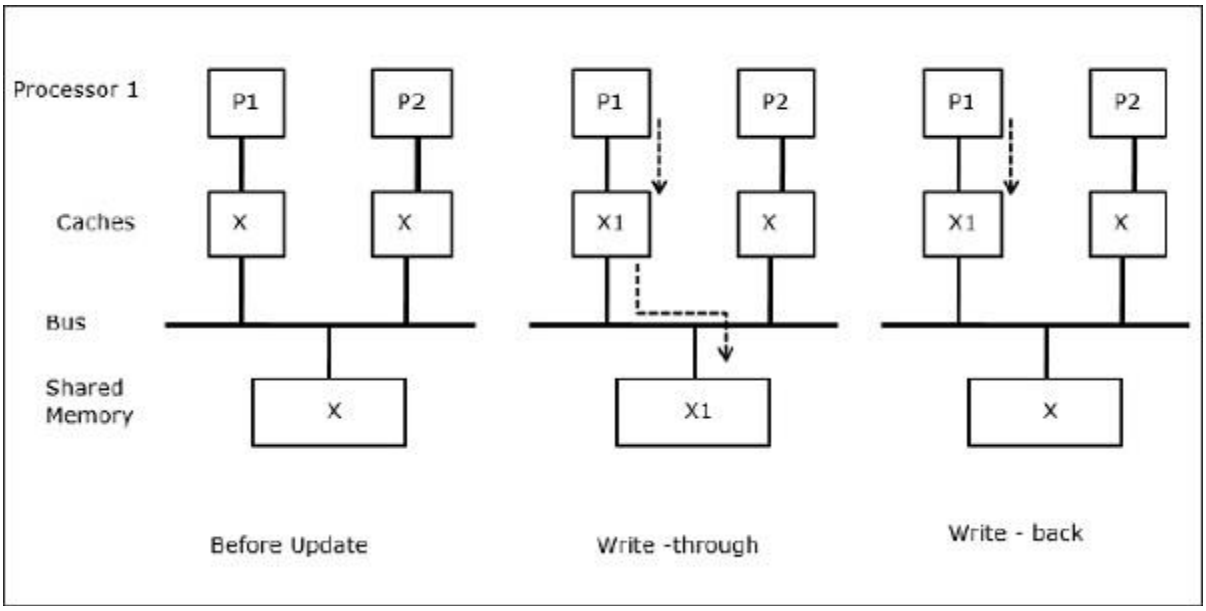
Maintaining cache coherency is a problem in multiprocessor system when the processors contain local cache memory. Data inconsistency between different caches easily occurs in this system.

The major concern areas are −

- Sharing of writable data
- Process migration
- I/O activity

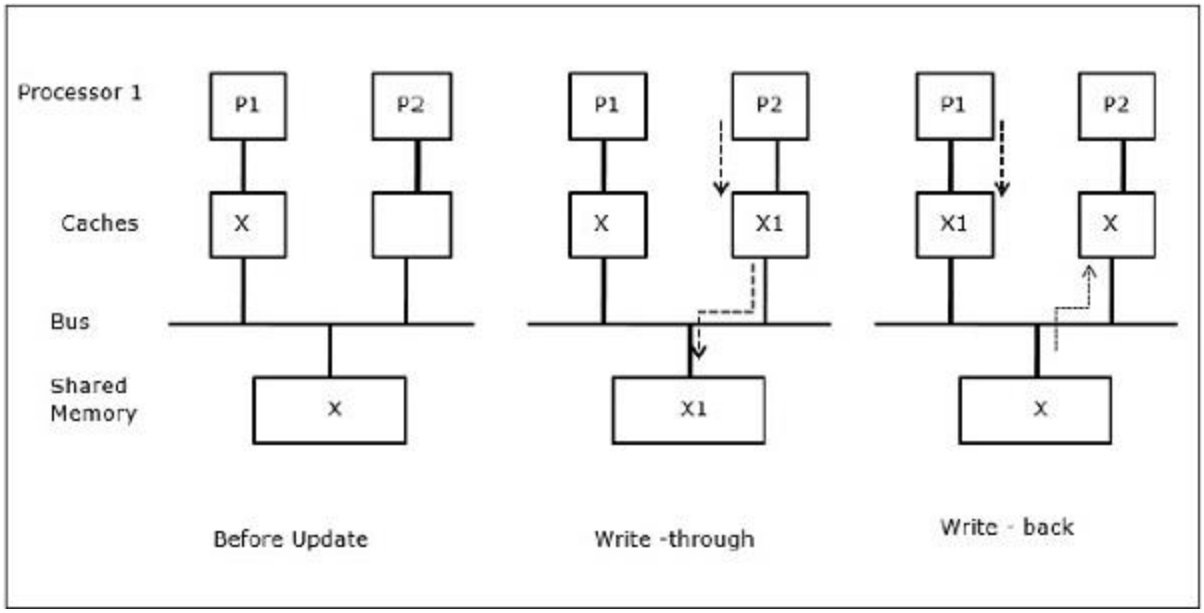
## Sharing of writable data

When two processors (P1 and P2) have same data element (X) in their local caches and one process (P1) writes to the data element (X), as the caches are write-through local cache of P1, the main memory is also updated. Now when P2 tries to read data element (X), it does not find X because the data element in the cache of P2 has become outdated.



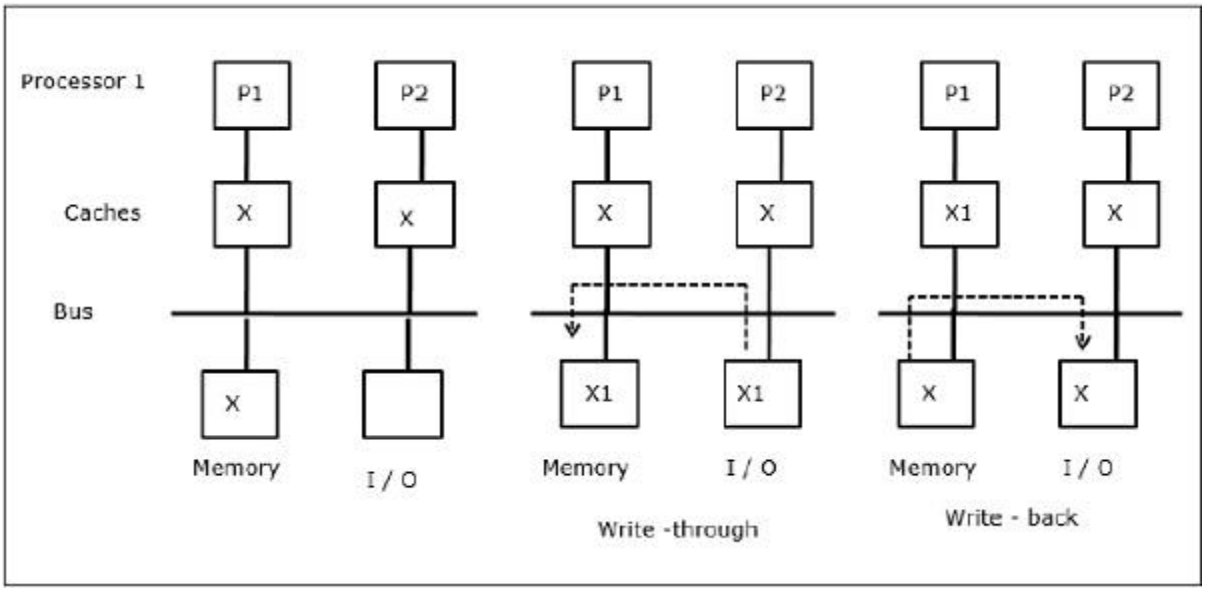

Process migration

In the first stage, cache of P1 has data element X, whereas P2 does not have anything. A process on P2 first writes on X and then migrates to P1. Now, the process starts reading data element X, but as the processor P1 has outdated data the process cannot read it. So, a process on P1 writes to the data element X and then migrates to P2. After migration, a process on P2 starts reading the data element X but it finds an outdated version of X in the main memory.




I/O activity

As illustrated in the figure, an I/O device is added to the bus in a two-processor multiprocessor architecture. In the beginning, both the caches contain the data element X. When the I/O device receives a new element X, it stores the new element directly in the main memory. Now, when either P1 or P2 (assume P1) tries to read element X it gets an outdated copy. So, P1 writes to element X. Now, if I/O device tries to transmit X it gets an outdated copy.




Uniform Memory Access (UMA)

Uniform Memory Access (UMA) architecture means the shared memory is the same for all processors in the system. Popular classes of UMA machines, which are commonly used for (file-) servers, are the so-called Symmetric Multiprocessors (SMPs). In an SMP, all system resources like memory, disks, other I/O devices, etc. are accessible by the processors in a uniform manner.

Non-Uniform Memory Access (NUMA)

In NUMA architecture, there are multiple SMP clusters having an internal indirect/shared network, which are connected in scalable message-passing network. So, NUMA architecture is logically shared physically distributed memory architecture.

In a NUMA machine, the cache-controller of a processor determines whether a memory reference is local to the SMP's memory or it is remote. To reduce the number of remote memory accesses, NUMA architectures usually apply caching processors that can cache the remote data. But when caches are involved, cache

coherency needs to be maintained. So these systems are also known as CC-NUMA (Cache Coherent NUMA).

## Cache Only Memory Architecture (COMA)

COMA machines are similar to NUMA machines, with the only difference that the main memories of COMA machines act as direct-mapped or set-associative caches. The data blocks are hashed to a location in the DRAM cache according to their addresses. Data that is fetched remotely is actually stored in the local main memory. Moreover, data blocks do not have a fixed home location, they can freely move throughout the system.

COMA architectures mostly have a hierarchical message-passing network. A switch in such a tree contains a directory with data elements as its sub-tree. Since data has no home location, it must be explicitly searched for. This means that a remote access requires a traversal along the switches in the tree to search their directories for the required data. So, if a switch in the network receives multiple requests from its subtree for the same data, it combines them into a single request which is sent to the parent of the switch. When the requested data returns, the switch sends multiple copies of it down its subtree.

## COMA versus CC-NUMA

Following are the differences between COMA and CC-NUMA.

- COMA tends to be more flexible than CC-NUMA because COMA transparently supports the migration and replication of data without the need of the OS.
- COMA machines are expensive and complex to build because they need non-standard memory management hardware and the coherency protocol is harder to implement.
- Remote accesses in COMA are often slower than those in CC-NUMA since the tree network needs to be traversed to find the data.