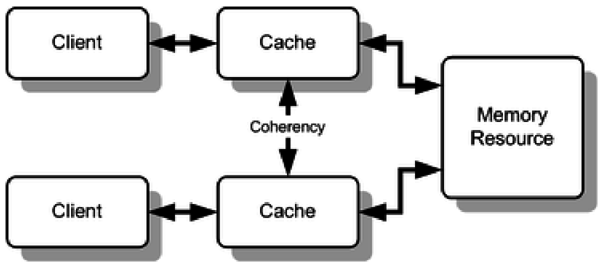
Cache coherence

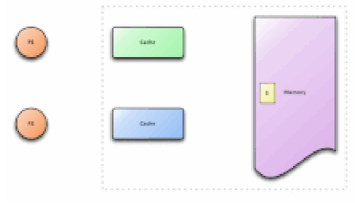
en.wikipedia.org/wiki/Cache_coherence

Jump to navigation Jump to search



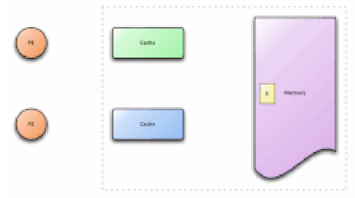
An illustration showing multiple caches of some memory, which acts as a shared resource

In computer architecture, **cache coherence** is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.



Incoherent caches: The caches have different values of a single address location.

In the illustration on the right, consider both the clients have a cached copy of a particular memory block from a previous read. Suppose the client on the bottom updates/changes that memory block, the client on the top could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.



Coherent caches: The value in all the caches' copies is the same.

Overview

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.[3]

Definition

Coherence defines the behavior of reads and writes to a single address location.[2]

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:[4]

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.

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.[5]

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".[3] 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".[6]

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,[7] snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached.[4] 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.[8]

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.[9]

Coherence protocols

Coherence protocols apply cache coherence in multiprocessor systems. The intention is that two clients must never see different values for the same shared data.

The protocol must implement the basic requirements for coherence. It can be tailor-made for the target system or application.

Protocols can also be classified as snoopy or directory-based. Typically, early systems used directory-based protocols where a directory would keep a track of the data being shared and the sharers. In snoopy protocols, the transaction requests (to read, write, or upgrade) are sent out to all processors. All processors snoop the request and respond appropriately.

Write propagation in snoopy protocols can be implemented by either of the following methods:

Write-invalidate
When a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy of the snooped memory location, which forces a read from main memory of the new value on its next access.[4]

Write-update
When a write operation is observed to a location that a cache has a copy of, the cache controller updates its own copy of the snooped memory location with the new data.

If the protocol design states that whenever any copy of the shared data is changed, all the other copies must be "updated" to reflect the change, then it is a write-update protocol. If the design states that a write to a cached copy by any processor requires other processors to discard or invalidate their cached copies, then it is a write-invalidate protocol.

However, scalability is one shortcoming of broadcast protocols.

Various models and protocols have been devised for maintaining coherence, such as MSI, MESI (aka Illinois), MOSI, MOESI, MERSI, MESIF, write-once, Synapse, Berkeley, Firefly and Dragon protocol.[1] In 2011, ARM Ltd proposed the AMBA 4 ACE[10] for handling coherency in SoCs.

See also

References

1. ^ Jump up to: *a* *b* *E. Thomadakis, Michael (2011). The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms (PDF). Texas A&M University. p. 30. Archived from the original (PDF) on 2014-08-11.*
2. ^ Jump up to: *a* *b* *Sorin, Daniel J.; Hill, Mark D.; Wood, David Allen (2011-01-01). A primer on memory consistency and cache coherence. Morgan & Claypool Publishers. OCLC 726930429.*
3. **^** Neupane, Mahesh (April 16, 2004). "Cache Coherence" (PDF). Archived from the original (PDF) on 20 June 2010.
4. **^** Rasmus Ulfsnes (June 2013). "Design of a Snoop Filter for Snoop-Based Cache Coherency Protocols" Archived 2014-02-01 at the Wayback Machine (PDF). *diva-portal.org*. Norwegian University of Science and Technology. Retrieved 2014-01-20.
5. **^** *Kriouile. Formal Analysis of the ACE Specification for Cache Coherent Systems-on-Chip. In Formal Methods for Industrial Critical Systems. Springer Berlin Heidelberg. ISBN 978-3-642-41010-9.*

Further reading

- *Patterson, David; Hennessy, John (2009). Computer Organization and Design (4th ed.). Morgan Kaufmann. ISBN 978-0-12-374493-7.*
- *Handy, Jim (1998). The Cache Memory Book (2nd ed.). Morgan Kaufmann. ISBN 9780123229809.*
- *Sorin, Daniel; Hill, Mark; Wood, David (2011). A Primer on Memory Consistency and Cache Coherence (PDF). Morgan and Claypool. ISBN 978-1608455645. Retrieved 20 October 2017.*

Parallel computing

General	• Distributed computing • Parallel computing • Massively parallel • Cloud computing • High-performance computing • Multiprocessing • Manycore processor • GPGPU • Computer network • Systolic array
Levels	• Bit • Instruction • Thread • Task • Data • Memory • Loop • Pipeline
Multithreading	• Temporal • Simultaneous (SMT) • Speculative (SpMT) • Preemptive • Cooperative • Clustered Multi-Thread (CMT) • Hardware scout

Theory	• PRAM model • PEM Model • Analysis of parallel algorithms • Amdahl's law • Gustafson's law • Cost efficiency • Karp–Flatt metric • Slowdown • Speedup
Elements	• Process • Thread • Fiber • Instruction window • Array data structure
Coordination	• Multiprocessing • Memory coherency • Cache coherency • Cache invalidation • Barrier • Synchronization • Application checkpointing
Programming	• Stream processing • Dataflow programming • Models ◦ Implicit parallelism ◦ Explicit parallelism ◦ Concurrency • Non-blocking algorithm
Hardware	• Flynn's taxonomy ◦ SISD ◦ SIMD ◦ SIMT ◦ MISD ◦ MIMD • Dataflow architecture • Pipelined processor • Superscalar processor • Vector processor • Multiprocessor ◦ symmetric ◦ asymmetric • Memory ◦ shared ◦ distributed ◦ distributed shared ◦ UMA ◦ NUMA ◦ COMA • Massively parallel computer • Computer cluster • Grid computer • Hardware acceleration

APIs	• Ateji PX • Boost • Chapel • HPX • Charm++ • Cilk • Coarray Fortran • CUDA • Dryad • C++ AMP • Global Arrays • GPUOpen • MPI • OpenMP • OpenCL • OpenHMPP • OpenACC • Parallel Extensions • PVM • POSIX Threads • RaftLib • UPC • TBB • ZPL
Problems	• Automatic parallelization • Deadlock • Deterministic algorithm • Embarrassingly parallel • Parallel slowdown • Race condition • Software lockout • Scalability • Starvation

Category: Parallel computing