Cache Coherence Techniques

Silvia Lametti

December 1, 2010

Master Program in Computer Science and Networking

Course High Performance Computing Systems and Enabling Platforms

Foreword

to the report "Cache Coherence Techniques" by Silvia Lametti

This survey report on cache coherence techniques is a part of the Master Thesis in Computer Science by Silvia Lametti, a new student of the PhD School in Computer Science of the University of Pisa.

The Master Thesis deals with a deep comparison of automatic and algorithm dependent approaches to cache coherence.

This excellent survey report covers, in a very clear and rigorous way, the set of techniques and protocols for automatic cache coherence, that can be considered the most interesting and widely applied at the current state-of-the-art of medium-low parallelism and of high-parallelism multiprocessor architectures.

This report is suggested as an optional teaching material of the High Performance Computing Systems and Enabling Platforms course, for the students interested to deepen this subject, and for scientific and/or professional culture.

Marco Vanneschi

Contents

1 Cache Coherence **4**

1.1 The Cache Coherence Problem 4

1.1.1 Invalidation vs Update 5

1.2 Cache Coherence Protocols 5

1.2.1 The MSI protocol 6

1.2.2 The MESI protocol 9

1.2.3 The MOSI protocol 12

1.2.4 The Dragon protocol 12

1.2.5 Multilevel cache hierarchies 15

2 Automatic Cache Coherence **18**

2.1 Snoopy-based 18

2.1.1 Snooping protocol for cache coherence 19

2.1.2 Cache-to-cache sharing 22

2.2 Directory-based 23

2.2.1 Flat 26

2.2.2 Hierarchical 35

2.3 Hybrid approaches 36

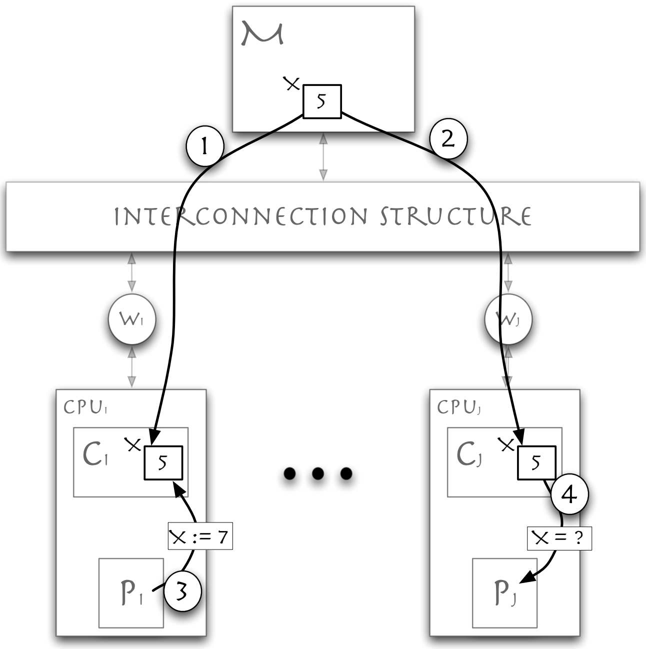


Figure 1: The cache coherence problem

1 Cache Coherence

In multiprocessor architectures caching plays a very important double role: as in uniprocessors, it contributes to minimize the instruction service time, but especially it contributes to reduce the memory congestion, that is the memory utilization factor.
In *all-cached* architectures, where any information is transferred into the primary cache before being used, the presence of writable shared information in caches introduces the problem of Cache Coherence.

1.1 The Cache Coherence Problem

The cache coherence problem arises from the possibility that more than one cache of the system may maintain a copy of the same memory block.
If different processors transfer into their cache the same block, it is necessary to ensure that copies remain consistent with each other and against the copy in main memory.

As summarized in figure 1, the system considered has a shared memory

M, it consists of processors P_i, each with its cache C_i. If P_i and P_j transfer the same variable X from the main memory M into their respective caches, if X is read-only no consistency problem arises. If P_i modifies X, the X copy in C_j becomes inconsistent. Moreover, in a write-back system, other processors find a inconsistent value of S in M, when they transfer X in their cache.

1.1.1 Invalidation vs Update

In many systems, the cache coherence techniques are entirely implemented at the firmware level. Two main techniques, called *automatic* cache coherence techniques, are used:

- *invalidation*, in which it is assumed that the only valid copy of a block is one of those, usually the last one, that has changed, invalidating all other copies; in a write-through system the copy in M is also valid;
- *update*, in which each modification of a cache block copy is communicated (multicasted) to all other caches.

In the previous example, if P_i wants use X, the first access will cause the transfer of the block which contains X in C_i. If P_j also has a copy of the same block in its cache, then the first of two, say P_i, which performs a write on the X block:

- with the invalidation mechanism, invalidates the copy of the block in C_j;
- with the update mechanism, updates the copy of the block in C_j by sending the updated data by diffusion.

At first sight, the update technique appears simpler, while invalidation is potentially affected by a sort of inefficient "ping-pong" effect. However, in the real utilization of cache coherent systems: the update mechanism has a substantially higher overhead, also due to the fact that only a small fraction of nodes contain the same block; on the other hand, processor synchronization reduces the "ping-pong" effect substantially, so invalidation is adopted by the majority of systems.

1.2 Cache Coherence Protocols

In all the systems which use the automatic techniques, a proper protocol must exists in order to perform the required actions *atomically.*
In a generic cache coherence protocol each block in a cache has a state associated with it, along with the tag and data, which indicates the disposition

of the block. The cache policy is defined by the *cache block state transition diagram*, which is a finite state machine specifying how the disposition of a block changes. While only blocks that are actually in cache lines have sate information, logically, all blocks that are not resident in the cache can be viewed as being in either a special "not present" state or in the "invalid" state.
In a uniprocessor system, for a write-through, write-no-allocate cache, only two states are required: valid and invalid. Initially, all the block are invalid; when a processor read causes a fault, the block is transferred from the memory into the cache and it's marked valid. Writes do not change the state of the block, they only update the memory and the cache block if it is present in the valid state. If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid.
A write-back cache requires an additional state per cache line, indicating a "dirty" or modified block.
In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block's cache state as being a set of n states, where n is the number of caches. The cache state is manipulated by a set of n distributed finite state machines, implemented by the units W of each node (see figure 1) that act as cache coherence controllers. The state machine that governs the state changes is the same for all blocks and all caches, but the current state of a block in different caches is different.

1.2.1 The MSI protocol

The MSI protocol is a basic invalidation-based protocol for write-back caches.
The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified from those that are modified (dirty). These states are:

- ***M**odified:* also called *dirty*, means that only this cache has a valid copy of the block, and the copy in main memory is stale;
- ***S**hared:* means the block is present in an unmodified state in this cache. main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy;
- ***I**nvalid:* has the obvious meaning.

Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated.

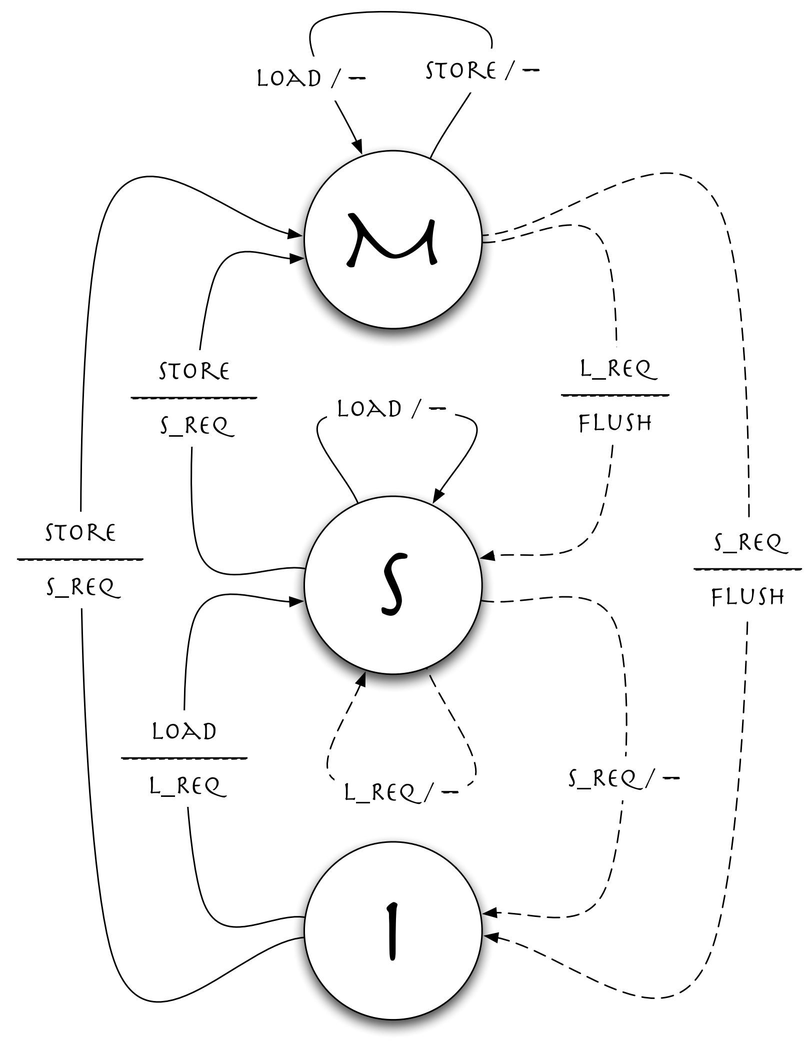


Figure 2: MSI State Diagram

The state transition diagram in figure 2 shows the possible state transitions of a cache block. All the diagram transitions have a label of the form R/A, where R indicates a request while A represent the action that the cache coherence controller must be taken following the request made. As in the others protocols, each of these transitions is composed of one or more operations and, for the correctness of the protocol, it's necessary that these be performed in an atomic way.
There are two types of transition, based on who makes the request:

- the *bold arcs* represent the transitions due to the read (`LOAD`) and write (`STORE`) processor issues;
- the *dashed arcs* represent the transition due to requests from the other caches (`L_REQ` and `S_REQ`).

The requests from the processor may refer to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in modified state, its contents will have to be written back to main memory (`WRITE_BACK`).

Transitions due to processor requests A processor read to a block that is invalid, or not present, involves sending a read request as a result of the fault. The newly loaded block is *promoted* from invalid to the shared state in the requesting cache. No action is taken if a read request refers to a block in a shared or modified state.
A processor write which refers to an invalid block cause a write fault and involves sending a write request. The newly loaded block is stored as a modified block. Instead, if the processor write refers to a shared block then, as before, the request cause a write fault in order to acquire the exclusive ownership. The data that comes back can be ignored in this case, since it is already in cache. In fact, a common optimization to reduce data traffic is to introduce a new type of transition that send *update* request which doesn't involves any data transfer. Additional writes to the block while it is in modified state generate no additional actions.
A replacement of a block from a cache logically corresponds to change the state to invalid; if the block being replaced was in modified state, the replacement transition from modified to invalid involves updating the main memory (`WRITE_BACK`). Instead, no action is taken if the block being replaced was in shared state.

Transitions due to requests from the other caches A read request from another cache to a block which is in cache in a modified state, involves executing of a particular operation (`FLUSH` of the data), that is sending the updated data from the cache to the requesting cache and the updating of main memory; after that the block state change to shared. No action is taken if the read request from another cache refers to a block which is in cache in a shared state.
A write request from another cache into a block which is in cache in a modified state, causes executing of the `FLUSH`, as in the read request, but in this case the block is invalidated. If the request refers to a block which is in cache in a shared state, the only action to be taken is the invalidation of the block.

1.2.2 The MESI protocol

The MESI protocol (known also as Illinois protocol due to its development at the University of Illinois at Urbana-Champaign [10]) is a widely used cache coherence protocol. It is the most common protocol which supports write-back cache.
Analyzing the MSI protocol, the first factor of inefficiency can be seen when a process needs to read and modify a data item: the transitions that are caused are always two, even when there are no other nodes sharing the cache block. In fact, it is initially generated a transition that gets the memory block in shared state; the second transition is caused by the processor write request that converts the state of the block from shared to modified.
The MESI protocol adds an ***Exclusive*** state to reduce the traffic caused by writes of blocks that only exist in one cache. This new state indicates an intermediate level of binding between shared and modified:

- unlike the shared state, the cache can perform a write and move to the modified state without further requests;
- it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block from another cache.

Transitions due to processor requests The state transition diagram in figure 3 shows the possible state transitions of a cache block caused by processor requests. When the block is first read by a processor, if a valid copy exists in another cache (condition (s) in figure), then it enters the processor's cache in shared state, as usual. However, if no other cache has a copy at the time (condition ($\bar{s}$) in figure), it enters the cache in exclusive state. The next read requests from the processor keep unchanged the status of the block.

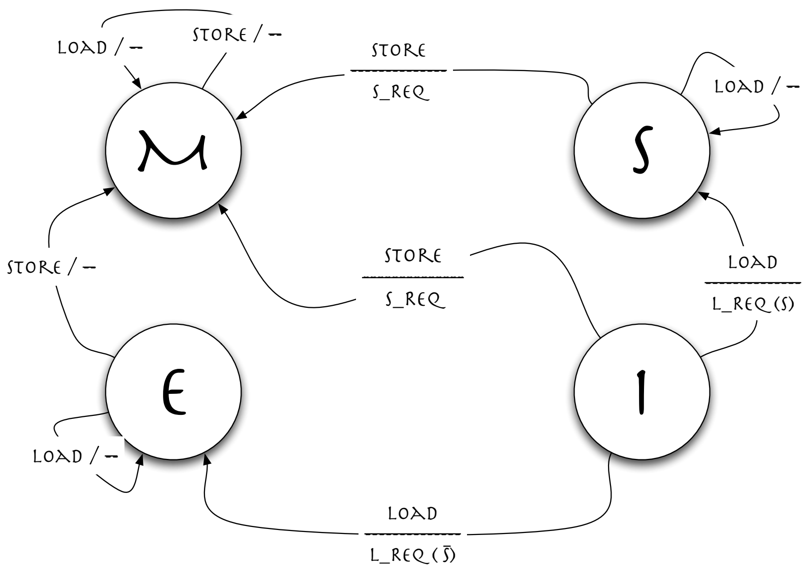


Figure 3: MESI state diagram: transitions due to processor requests

When the block is written by the same processor, it can directly transition from exclusive to modified state without generating another request since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been demoted from exclusive to shared by the protocol.

Transitions due to requests from the other caches The state transition diagram in figure 4 shows the possible state transitions of a cache block caused by requests from the other caches.
A read request from another cache for an exclusive block causes a state transition from exclusive to shared.
A write request form another cache causes the invalidation of the block.
With the introduction of the exclusive state, in multiprocessor with *Snoopy-based* cache coherence protocol (see section 2.1), a technique called *cache-to-cache sharing* can be used for the sharing of blocks between caches. The `FLUSH` operations shown in parentheses refer to the use of this technique.

The Write Once Protocol

A write may only be performed if the cache line is in modified or exclusive state. If it is in the shared state, all other cached copies must be invalidated

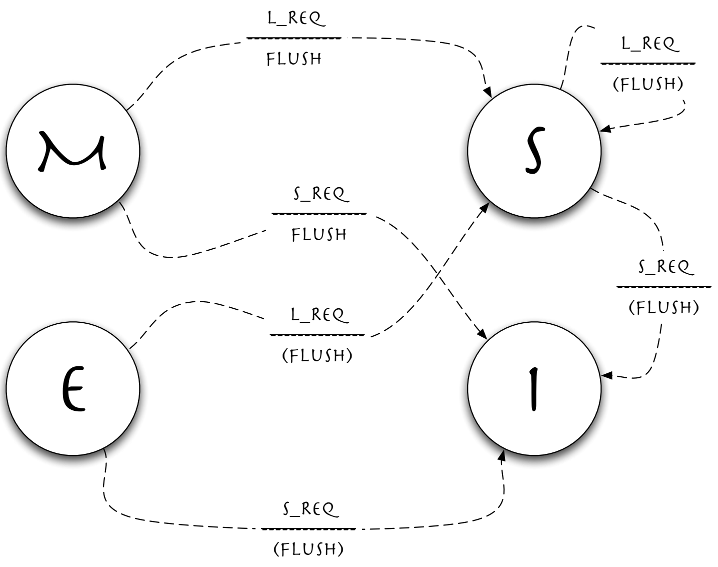


Figure 4: MESI state diagram: transitions due to requests from the other caches

first. This is typically done by a broadcast operation known as *Read For Ownership (RFO)*. In cache coherence protocol literature, Write-Once is the first *write-invalidate* protocol defined.
In this protocol, each block in the local cache is in one of these four states:

- Invalid;
- Valid;
- Reserved;
- Dirty.

These states have exactly the same meanings as the four states of the MESI protocol (they are simply listed in reverse order), but this is a simplified form of it that avoids the Read for Ownership operation. Instead, all invalidation is done by writes to main memory. In particular, the first time that the processor intends to write in a valid block (shared) uses the technique of write-through, which implicitly invalidates all other copies of the block. At this point, the block is cached in a reserved state (exclusive), and subsequent writes may be carried out simply by changing the status of the cache block in dirty (modified).

1.2.3 The MOSI protocol

The MOSI protocol is another extension of the basic MSI cache coherence protocol. It adds the ***Owned*** state. A cache line in the owned state holds the most recent, correct copy of the data. This state is:

- similar to the shared state in that other processors can hold a copy of the most recent, correct data;
- similar to the modified state, in fact the copy in main memory can be stale (incorrect).

Only one cache can hold the data in owned state, all other caches must hold the data in shared state. The state block may be changed to modified after invalidating all shared copies, or changed to shared by writing the modifications back to main memory.
In response to requests for read and/or write by another cache for an owned block, the cache must performs the `FLUSH` of the data, such as the modified state; in this case, however, the request for writing does not necessarily entail the invalidation of the block, which can be kept in cache in shared state.

The MOESI protocol

The MOESI protocol, introduced in [11], encompasses all of the possible states commonly used in other protocols. Figure 5 allows us to understand how to classify a cache block with the MOESI protocol, according to the following three characteristics:

- validity;
- exclusiveness;
- ownership.

This avoids the need to write modified data back to main memory before sharing it. While the data must still be written back eventually, the write-back may be deferred.

1.2.4 The Dragon protocol

Let us examine a basic update-based protocol for write-back caches. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system [9]. The Dragon protocol consists of four states:

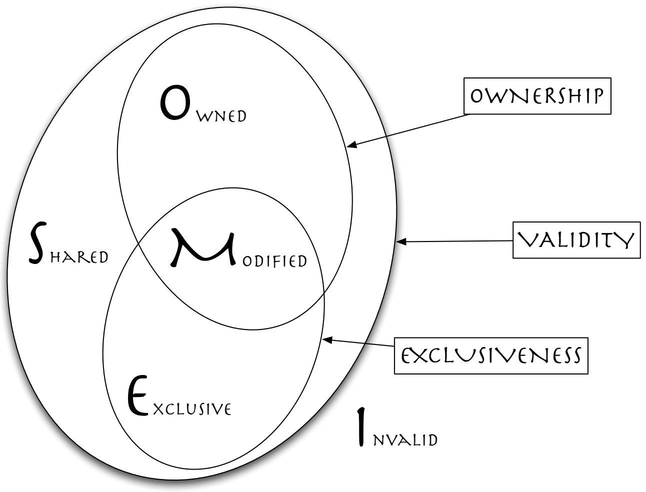


Figure 5: MOESI classification

- ***E**xclusive-clean:* (or exclusive) has the same meaning and the same motivation as before: only one cache (this cache) has a copy of the block, and it has not been modified (the main memory is up-to-date);
- ***S**hared-**c**lean:* means that potentially two or more caches (including this one) have this block, and main memory may or may not be up-to-date;
- ***S**hared-**m**odified:* means that potentially two or more caches have this block, main memory is not up-to-date, and it is this cache's responsibility to update the main memory at the time this block is replaced from the cache; a block may be in this state in only one cache at a time; however it is quite possible that one cache has the block in this state, while others have it in shared-clean state;
- ***M**odified:* signifies exclusive ownership as before; the block is modified and present in this cache alone, main memory is stale, and it is this cache's responsibility to supply the data and to update main memory on replacement.

Note that there is no explicit invalid state as in the previous protocols, because it is an update-based protocol. The protocol always keeps the blocks

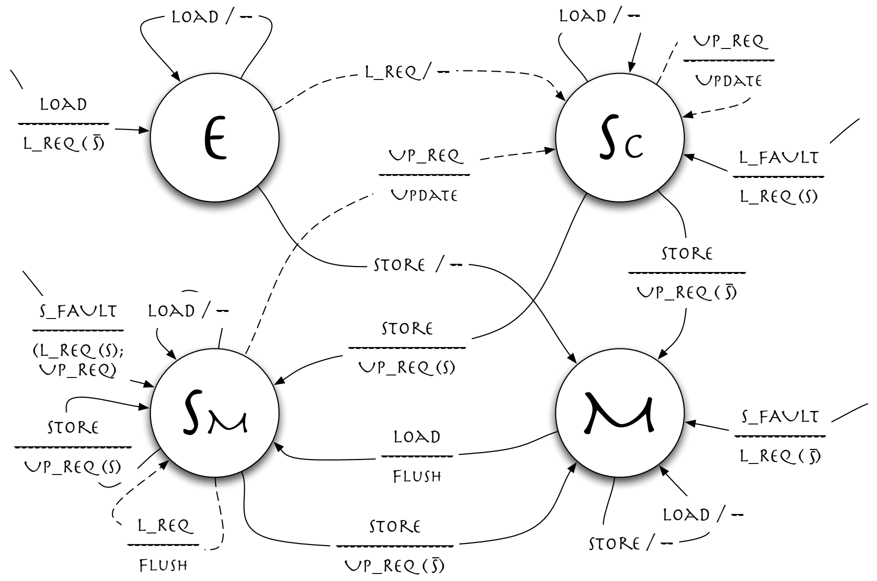


Figure 6: Dragon protocol

in the cache up-to-date, so it is always okay to use the data present in the cache if the tag match succeeds. However, if a block is not present in a cache at all, it can be imagined in a special invalid or not-present state.
Figure 6 shows the state transition diagram of the Dragon protocol. Since we do not have an invalid state, to specify actions it is necessary to distinguish whether the requested block is present or not in cache. In addition to the read requests (`LOAD`) and writing (`STORE`) of a block in this cache, we also consider the generation of fault as a result of cache reads (`L_FAULT`) and writing (`S_FAULT`).
The actions that the cache coherence controller can be taken in response to requests of the node are: the read (`L_REQ`) and the update (`UP_REQ`) requests; can also be performed the write-back (`WRITE_BACK`) and the update (`UPDATE`) of the block. The update request concerns a specific word of the block and is used to maintain updated the caches that share the block. To do this, each controller uses the `UPDATE` action to keep updated the local copy.
Let's look at what action is taken when a cache incurs a fault for a read request, a write (which causes a fault or not), or a replacement.

Fault for a read request A read request is generated. Depending on the status, that is if the block is in another cache (s) or not ($\bar{s}$), the block is loaded in exclusive or shared-clean state in the local cache. If the block is in modified or shared-modified in one of the other caches, that cache supplies the latest data for that block, and the block is loaded in the local cache in shared-clean state. If the block is in shared-clean state in other caches, memory supplies the data, and it is loaded in shared-clean state. If no other cache has a copy, the data is supplied by the main memory, and the block is loaded in the local cache in exclusive state.

Write If the block is in modified state in the local cache, then no action needs to be taken. If the block is in the exclusive state in the local cache, then it changes to modified state and again no further action is needed. If the block is in shared-clean or shared-modified state, however, an update request is sent. If any other caches have a copy of the data, they update their cached copies and change their state to shared-clean if necessary. The local cache also updates its copy of the block and changes its state to shared-modified if necessary. If no other cache has a copy of the data, the local copy is updated and the state is changed to modified.
Finally, if on a write the block is not present in the cache, the write is treated simply as a fault for a read request: a read request is generated followed by a write request. If the block is also found in other caches, an update request is generated, and the block is loaded locally in the shared-modified state; otherwise, the block is loaded locally in the modified state.

Replacement On a replacement, the block is written back to memory only if it is in modified or shared-modified state. If it is in the shared-clean state, then either some other cache has it in shared-modified state or none does, in which case it is already valid in main memory.

1.2.5 Multilevel cache hierarchies

The simple design presented in the preceding section was illustrative, but it made a simplifying assumption that is not valid on most modern systems: single-level caches. Many systems use on-chip secondary caches as well and an off-chip tertiary cache. Multilevel cache hierarchies would seem to complicate coherence since changes made by the processor to the first-level cache may not be visible to the cache coherence controller and, in a snoopy-based system bus transactions are not directly visible to the first-level cache.
Let us consider a two-level hierarchy for concreteness; the extension to the multilevel case is straightforward. One obvious way to handle multilevel caches is to have independent cache coherence controller for each level of the cache hierarchy. This is unattractive for several reasons. In addition to making complicated the system design, most of the time, the blocks present in

the L_1 cache are also present in the L_2 cache; therefore, the communications between the levels (the snoop, in the snoopy-based systems) are unnecessary. The solution used in practice is based on this last observation. When using multilevel caches, designers ensure that they preserve the *inclusion property*, which requires the following:

- if a memory block is in the L_1 cache, then it must also be present in the L_2 cache; in other words, the contents of the L_1 cache must be a subset of the contents of the L_2 cache;
- if the block is in an owned state (e.g., modified in MESI, owned in MOSI, shared-modified in Dragon) in the L_1 cache, then it must also be marked modified in the L_2 cache.

The first requirement ensures that all actions taken that are relevant to the L_1 cache are also relevant to the L_2 cache, so having only a cache coherence controller for the L_2 cache is sufficient. The second ensures that if a request for a block that is in modified state in the L_1 cache or L_2 cache, then it is enough to keep track of this information only for the L_2 cache.

Maintaining inclusion

The requirements for inclusion are not trivial to maintain. Three aspects need to be considered:

- processor references to the L_1 cache cause it to change state and perform replacements; these need to be handled in a manner that maintains inclusion;
- requests from other nodes cause the L_2 cache to change state and flush blocks; these need to be forwarded to the first level;
- the modified state must be propagated out to the L_2 cache.

At first glance, it might appear that inclusion would be satisfied automatically since all L_1 cache fault go to the L_2 cache. The problem is that the implementation of this approach can be complicated by the use of certain techniques typically implemented in cache hierarchies, such as block replacement policies based on the history of access (e.g., LRU replacement policy), the use of more cache at the same level (e.g., first-level cache is divided into instruction cache and data cache) or the use of different block sizes (σ_1 and σ_2) between the two levels of hierarchy. In order not to give up the benefits obtained from the use of these techniques, inclusion is maintained explicitly

by extending the mechanisms used for propagating coherence events through the cache hierarchy.
Whenever a block in the L_2 cache is replaced, the address of that block is sent to the L_1 cache, asking it to invalidate or flush (if modified) the corresponding blocks (there can be multiple blocks if $\sigma_2 > \sigma_1$).
Considering requests from other nodes, some, but not all, of these requests relevant to the L_2 cache are also relevant to the L_1 cache and must be propagated to it. For example, if a block is invalidated in the L_2 cache, the invalidation must also be propagated to the L_1 cache if the data is present in it. Inform the L_1 cache of all actions that were relevant to the L_2 cache is the easiest solution but in many cases it is useless. A more attractive solution is for the L_2 cache to keep extra state (inclusion bit) with cache blocks; which record whether the block is also present in the L_1 cache. It can then suitably filter interventions to the L_1 cache at the cost of a little extra hardware and complexity.
Finally, on an L_1 write, the modifications needs to be communicated to the L_2 cache so it can supply the most recent data if necessary. One solution is to make the L_1 cache write-through. The requirement can also be satisfied with write-back L_1 caches since it is not necessary that the data in the L_2 cache be up-to-date but only that the L_2 cache knows when the L_1 cache has more recent data. Thus, the state information for L_2 cache blocks is augmented so that blocks can be marked "modified-but-stale". The block in the L_2 caches behaves as a modified block for the cache coherence protocol, but data is fetched from the L_1 cache when it need to be flushed to other nodes. One simple approach is to set both the modified and invalid bits in the L_2 cache.
Keep the inclusion property allows for advantages in performance, even in systems where a level of cache hierarchy is shared (as shown in [6]), as it avoids, as also said before, making unnecessary communications.
Other techniques for the maintenance of the inclusion property are presented in [2].

2 Automatic Cache Coherence

Automatic cache coherence techniques allow programmers to develop programs without taking into account the cache coherence problem. In fact no explicit coherence operations must be inserted in the program.
Two main classes of architectural solutions have been developed for automatic caching:

- *Snoopy-based*, described in section 2.1, in which a bus is used as a centralization point at firmware level;
- *Directory-based*, which implements cache coherence protocols using shared data in main memory, as explained in section 2.2; this solution is adopted in highly parallel systems, with powerful interconnection networks.

2.1 Snoopy-based

A simple solution to the problem of cache coherence is one that uses a firmware level point of centralization, in particular the use of a bus, also known as *Snoopy bus.* Each of devices connected to the bus can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, the unit W, which act as cache controller, examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having all cache controllers "snoop" on the bus and monitor the transactions from other nodes, as illustrated in figure 7. The unit W may take action if a bus transaction is relevant to it, that is, if it involves a memory block of which it has a copy in its cache. The key properties of a bus that support coherence are the following:

- all transactions that appear on the bus are visible to all cache controllers;
- they are visible to all controllers in the same order, the order in which they appear on the bus.

The check to determine if a bus transaction is relevant to a cache is essentially the same tag match that is performed for a request from the processor. The action taken may involve invalidating or updating the contents or state of that cache block and/or supplying the latest value for that block from the cache to the bus.

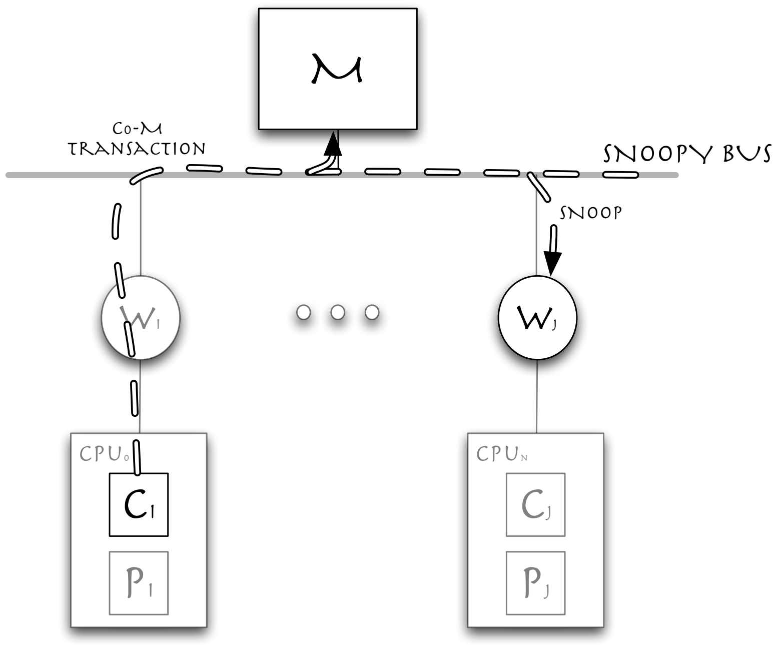


Figure 7: Cache Coherence through Bus Snooping

2.1.1 Snooping protocol for cache coherence

In a snooping cache coherence system, each cache controller receives two sets of inputs:

- the processor issues memory requests;
- the bus snooper informs about bus transactions from other caches.

In response to either, the unit W may update the state of the appropriate block in the cache according to the current state and the state transition diagram.
Thus, a snooping protocol is a distributed algorithm represented by a collection of cooperating finite state machines, which is specified by the following components:

- the set of states associated with memory blocks in the local caches;
- the state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction and produces as output the next state for the cache block;

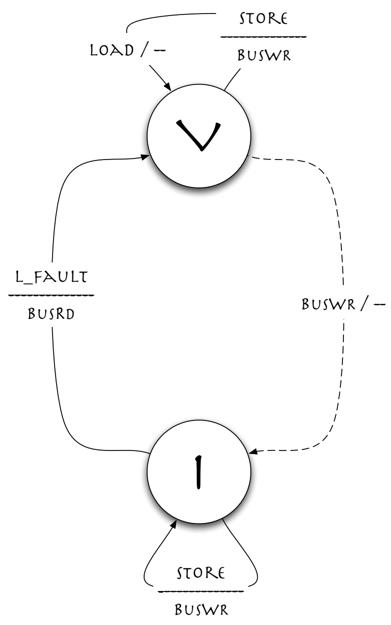


Figure 8: Snoopy coherence for a multiprocessor with write-through, write-no-allocate caches

- the actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, the cache and the processor.

As introduced in section 1, the set of operations required to execute a state transition must be performed in an atomic way. In this particular case, atomicity is achieved merely by the bus (*atomic bus*); in fact, only one transaction is in progress on the bus at a time.
Let's see now how the units W interact with the bus in order to maintain consistency. A simple invalidation-based protocol for a coherent write-through, write-no-allocate cache is described by the state transition diagram in figure 8. In particular, suppose that the bus makes available the following transactions:

- *BusRd* for the read request, which includes the address of the requested data;

- *BusWr* for the write request, which, in addition to the address, including the data to write;

As shown in the figure, when the unit W sees a read request from its processor which is referred to data not present in cache, a BusRd transaction is generated, and upon completion of this transaction the block transitions up to the valid state. Whenever the controller sees a processor write request for a location, a bus transaction is generated that updates that location in main memory with no change of state. If any processor generates a write request for a block that is cached in this cache, then the block must be invalidated.

Maintaining coherence in the write-through protocol In the write-through protocol all writes appear on the bus, feature that greatly simplifies the maintenance of coherence. In fact, since only one bus transaction is in progress at a time, in any execution all writes to a location are serialized by the order in which they appear on the bus. Since each cache controller performs the invalidation during the bus transaction, invalidations are performed by all cache controllers in that order. In this way, the protocol imposes a partial ordering of operations, from which can be construct a hypothetical total order that allows the maintenance of cache coherence. In particular the order can be formalized as follows:

- a memory operation M_1 is subsequent to a memory operation M_2 if the operations are issued by the same processors in the same order;
- a read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W;
- a write operation is subsequent to a read or write operation M if M generates a bus transaction and the bus transaction for the write follows that for M;
- a write operation is subsequent to a read operation if the read does not generate a bus transaction (it refers to a block already present in cache) and is not already separated from the write by another bus transaction.

As we said, this sort can be established easily because each `STORE` instruction is immediately translated into a BusWr transaction.
Let's analyze now what happens in the protocols for write-back caches. As introduced in section 1.2, write-back caches cause the introduction of the modified state. This results in the exclusive possession of the block; exclusivity, which means that the cache may modify the block without notifying the changes to other nodes.
To accomplish this the bus must provide a particular type of transaction called *read exclusive (BusRdX)*, sent from the unit W on the by when it receive a write request from the processor to a block that is:

- either not in cache;

- or is in the cache but not in the modified state.

When the unit W observes a BusRdX transaction on the bus, it must determine whether the requested block is present in the cache and, in this case, it must invalidate its copy.
Another type of transaction required in the case of write-back cache: the *BusWB* transaction is generated by a cache controller on a write back. This transaction is also generated by the `FLUSH` operation after a request, from another node, for a block that is in cache in the modified state.
In the MESI protocol, introduced in section 1.2.2, but also in the Dragon protocol (see section 1.2.4, the introduction of the concept of sharing a block leads to a distinction in the state change of a block, a distinction based on the fact that the block is in the shared state in another cache or not. In protocols that use a Snoopy Bus is necessary that the interconnection structure provides a mechanism to determine when this condition occurs. To do this the bus offer an additional signal, called *shared signal (S)*. When a cache controller observes a transaction and determines if the block is in its cache, if so it must send the shared signal that is put in OR with all the signals from other nodes. In this way the unit W which sent the transaction can determine if the block is present in another cache in the shared state.

Maintaining coherence in the write-back protocol As we have seen, the BusRdX transaction ensures that the cache which is writing into a block has the only valid copy, just like with the BusWr in the write-through cache. Although not all writes generate bus transaction, we know that between two transactions that relate to a block, that the processor P_i has obtained as exclusive copy, only P_i can execute write operation on this block. When another processor P_j request a read for the block, we know that there is at least one transaction on the bus (generated by the `FLUSH` operation) for that block that separates the completion of the P_i write operation by the read operation request by P_j. So this transaction ensures that all read operations see the effects of previous write operation.

2.1.2 Cache-to-cache sharing

An interesting issue that arises with the use of automatic techniques based on snooping is the choice of who should be charged for sending the data after the observation of a BusRd or a BusRdX transaction where the block is updated in main memory and in its cache. With the introduction of the *exclusive* state in the MESI protocol (see section 1.2.2) in fact, if the cache has a block in the *shared* state, and a read or a write request from another cache refers to it, then the cache may or may not execute the `FLUSH` operation, this choice depends on whether or not it enabled cache-to-cache sharing. Figure 4 shows that the `FLUSH` operation is optional (brackets in the image). This is because it can sometimes be more convenient, in terms of transfer times, which is responsibility of one of the cache, that have a valid copy of block data, send data to the requester.

The MESIF protocol When more than one cache has a valid copy of the block, it is necessary to have a selection algorithm to determine which of these should provide the data. The MESIF protocol was developed by Intel, and explained in [3], to solve this problem.
To do this the protocol use a new state ***Forward***, which indicates that the cache should act as a designated responder for any requests for the block.

2.2 Directory-based

In highly parallel systems are used interconnection structures that allow greater scalability than what can be achieved with linear latency networks. This choice is also reflected in the decision to integrate automatic cache coherence mechanisms that scale better than the solutions based on Snoopy bus. In fact, the protocols based on the snooping technique require that each node, including any unit W, which acts as a controller of consistency, can communicate with every other node in order to implement the protocol. To indicate this type of architecture, typical with a NUMA organization, which provide a primitive and scalable support for cache coherence, we use the term *CC-NUMA (Cache-Coherent, Non-Uniform Memory Access)*
Scalable cache coherence is typically based on the concept of a *directory*. Since the state of a block in the caches can no longer be determinated implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place, called directory. Imagine that each memory block corresponding to a cache block has associated with it a record of the caches that currently contain a copy of the block and the state of the block in those caches. This record is called the *directory entry* for that block, as shown in figure 9. As in Snoopy-based techniques, there may be many caches with a clean, readable block, but if the block is writable (possibly modified) in one cache, then only that cache may have a valid copy. When the node generate a fault for a block, it first communicates with the directory for the block, then determine, according to information in the directory entry, where the valid cached copies (if any) are and what further actions to take. t may take further communications from the node to maintain the consistency of the block. All access to directories

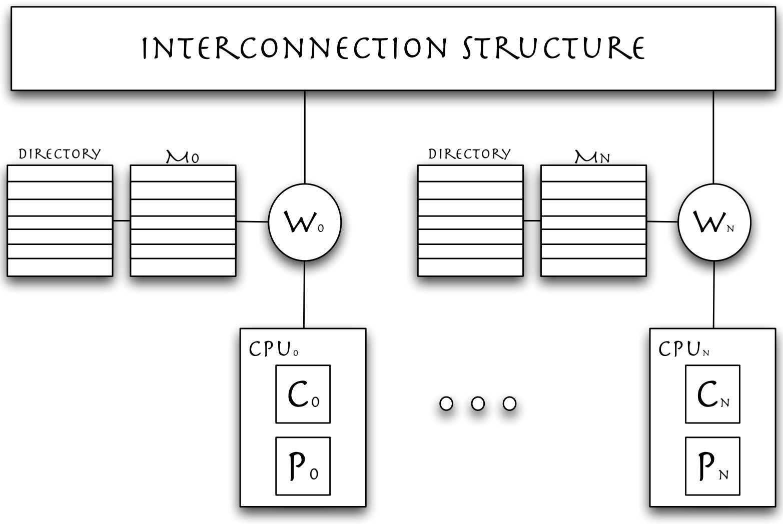


Figure 9: A NUMA organization with directories

and communications between nodes to maintain cache coherence are managed by the unit W, which acts as a controller of consistency.
By their nature, directory-based mechanisms are independent of the interconnection structure used.
The cache coherence protocol uses in directory-based systems may be invalidation based, update based, or hybrid, and the set of the cache states are very often the same, typically that of the MESI protocol (see section 1.2.2).
Given a protocol, a coherent system must provide mechanisms for managing the protocol; in particular it must perform the following steps when an access fault occurs:

1. finding out enough information about the state of the location (cache block) in other caches to determine what action to take;
2. locating those other copies, if needed (e.g., to invalidate them);
3. communicating with the other copies (e.g., obtaining data from them or invalidating or updating them).

In snoopy-based protocol, all these operations are performed by the broadcast and snooping mechanism. In directory-based protocol instead, information about the state of blocks in other caches is found by looking up the directory, while the location of the copies and any communications between the nodes

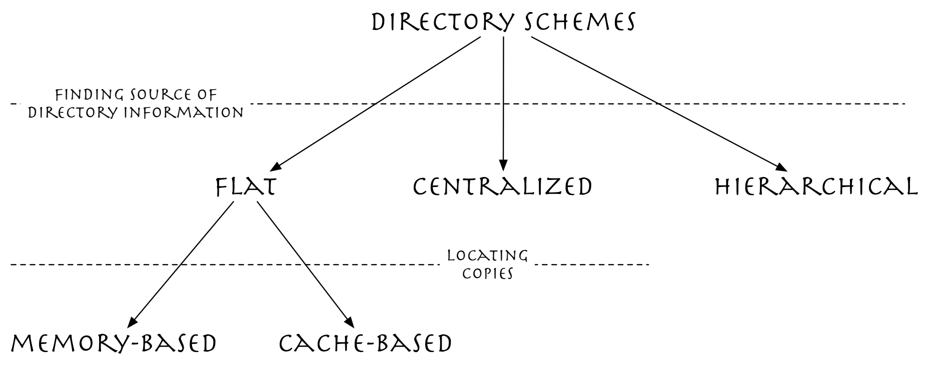


Figure 10: Directory-based schemes classification

are done through interprocessor communications between nodes, without resorting to broadcast communications.
Since communication with cached copies is always done through interprocessor communications, the real differentiation among directory-based approaches is in the first two operations of cache coherence protocols: finding the source of the directory information and determining the locations of the relevant copies. The first operation allows to identify two possible schemes that are a great alternative to a centralized solution: the simplest is the *flat* scheme, introduced in section 2.2.1, while the *hierarchical* scheme is described in section 2.2.2. Flat schemes can be divided into two classes, according to the location of the other copies of the block: memory-based and cache-based approaches. Figure 10 summarizes this classification.

The following definitions are useful to distinguish the processing nodes that interact with one another; for a given cache block:

- *home*: is the node in whose main memory the block is allocated;
- *local* (or requestor): is the node that issues a request for the block;
- *dirty*: is the node that has a copy of the block in its cache in modified (dirty) state; note that the home node and the dirty node for a block may be the same;
- *owner*: is the node that currently holds the valid copy of a block and must supply the data when needed; in directory protocols, this is either the home node (when the block is not in dirty state in a cache) or the dirty node;

- *exclusive*: is the node that has a copy of the block in its cache in an exclusive state, either dirty or (clean) exclusive as the case may be; thus, the dirty node is also an exclusive node.

2.2.1 Flat

Flat schemes are so named because the source of the directory information for a block is in a fixed place, usually at the home that is determined from the address of the block. On a fault, a single request is sent directly to the home node to look up the directory (if the home node is remote). Flat schemes can be divided into two main classes, depending on where can be obtained all the information relating to a block:

- *memory-based* schemes, that store the directory information about all cached copies at the home node of the block;
- *cache-based* schemes, where the information about cached copies is not all contained at the home but is distributed among the copies themselves; the home simply contains a pointer to one cached copy of the block; each cached copy then contains a pointer to the node that has the next cached copy of the block, in a distributed linked list organization.

Memory-based

Consider the use of a memory-based scheme: as shown in Figure 11, the directory information is kept together with the main memory of each node.

When a cache fault occurs, the local node sends a request to the home node where the directory information for the block is located.
On a read fault, the directory indicates from which node the data may be obtained, as shown in figure 12. On a write fault, the directory identifies the copies of the block, and invalidation request may be sent to these copies, as shown in figure 13. Recall that a write to a block in shared state is also considered as a write fault.

A simple organization for the directory information for a block is as a bit vector of $Npresencebits$, which indicate for each of the N nodes, whether that node has a cached copy of the block, together with one or more state bits. Let us assume for simplicity that there is only one state bit, called the *dirty bit*, which indicates if the block is dirty in one of the node caches. Of course, if the dirty bit is TRUE, then only one node (the dirty node) should be caching that block and only that node's presence bit should be TRUE.

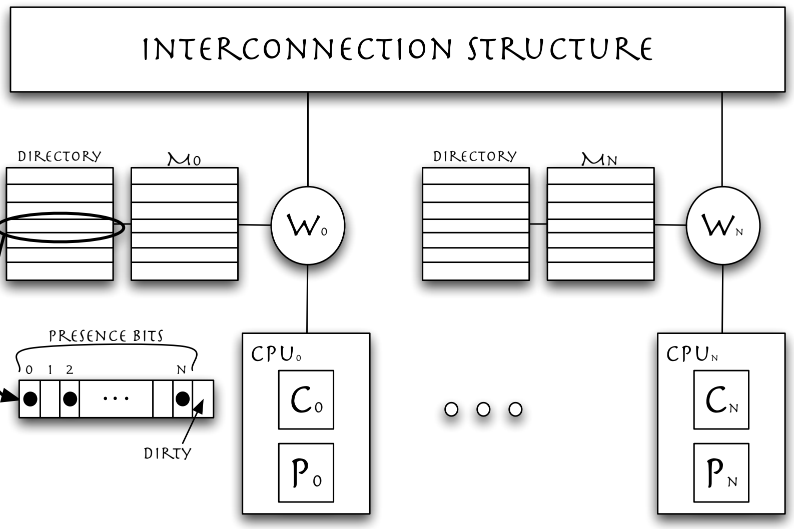
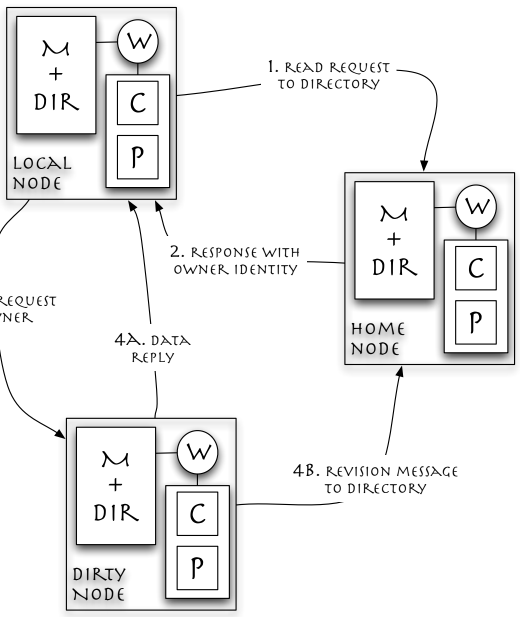


Figure 11: Directory memory-based

The directory information for a bock is simply main memory's view of the cache state of a block in different caches; the directory does not necessarily need to know the exact state (e.g., MESI) in each cache but only enough information to determine what actions to take.
To see in greater detail how a cache fault might interact with this bit vector directory organization, consider a protocol with three stable cache states (MSI), a single level of cache and uniprocessor nodes.
The protocol is orchestrated by the unit W of each node, which acts as cache controller. On a cache fault at node i, the unit W_i looks up the address of the memory block to determine if the home is local or remote. If it is remote, a request is sent through the interconnection structure to the home node for the block, suppose to be the node j. There, the the directory entry for the block is looked up, and W_j may treat the fault.

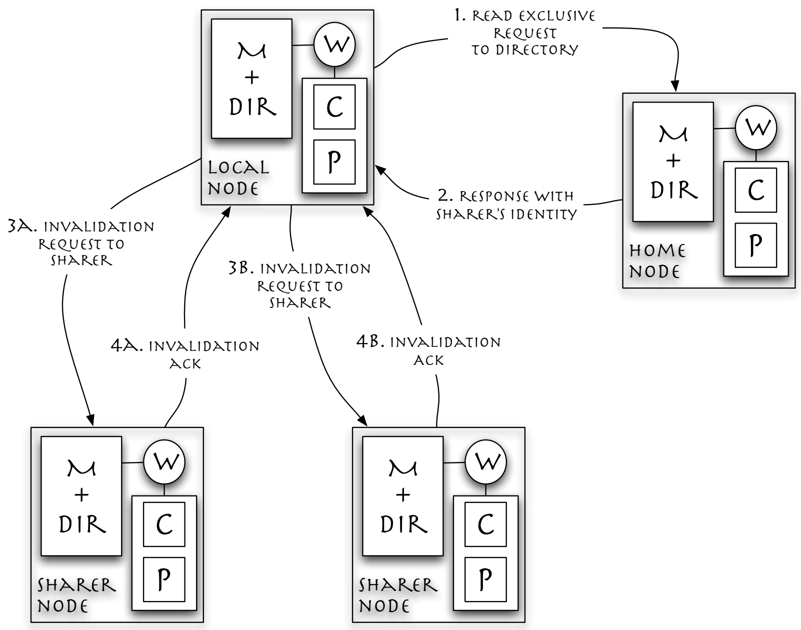
If the fault is caused by a read operation:

- if the dirty bit is FALSE, then W_j obtains the block from the main memory M_j, send the data to the node i and sets the ith presence bit to TRUE;
- if the dirty bit is TRUE, then W_j send to the node i the identify of the node k whose presence bit is TRUE (i.e., the owner or dirty node); W_j can send a block transfer request to the node k, which must change



M + DIR
W
C
P
LOCAL NODE
1. READ REQUEST TO DIRECTORY
HOME NODE
2. RESPONSE WITH OWNER IDENTITY
3. READ REQUEST TO OWNER
4A. DATA REPLY
DIRTY NODE
4B. REVISION MESSAGE TO DIRECTORY

Figure 12: Read fault to a block in modified state in a cache



M + DIR
W
C
P
LOCAL NODE
1. READ EXCLUSIVE REQUEST TO DIRECTORY
HOME NODE
2. RESPONSE WITH SHARER'S IDENTITY
3A. INVALIDATION REQUEST TO SHARER
3B. INVALIDATION REQUEST TO SHARER
4A. INVALIDATION ACK
4B. INVALIDATION ACK
SHARER NODE
SHARER NODE

Figure 13: Write fault to a block with two sharers

the block state to shared in its cache and send the data to both nodes i and j; node i stores the block in its cache in shared state, while node j stores the block in the main memory, where the dirty bit is reset and the ith presence bit is set to TRUE.

If the fault is caused by a write operation:

- if the dirty bit is FALSE, then main memory has a clean copy of the data; invalidation request must be sent to all nodes j for which the jth presence bit is TRUE; to do this, the home node supplies the block to the requesting node i together with the presence bit vector; then it resets the directory entry, leaving only the ith presence bit and the dirty bit with a TRUE value (if the request is an upgrade instead of a read exclusive, an acknowledgment containing the bit vector is returned to the requestor instead of the data itself); at this point, W_i sends invalidation requests to the required nodes and waits for invalidation acknowledgment from the nodes; finally, the node i places the block in its cache in modified state;

- if the dirty bit is TRUE, then the block is first recalled from the node j, whose presence bit is TRUE; that cache changes its state to invalid, and then the block supplied to the node j, which places the block in its cache in modified state; the directory entry is cleared, leaving only the ith presence bit set on TRUE.

On a replacement of a dirty block by node i, the dirty data being replaced is *written back* to main memory, and the directory is updated to reset the dirty bit and the ith presence bit. Finally, if a block in shared state is replaced from a cache, a message may or may not be sent to the directory to reset the corresponding presence bit so an invalidation is not sent to this node the next time the block is written.
The bit vector organization described earlier, called a *full bit vector* organization, is the most straightforward way to store directory information in a flat, memory-based scheme. The main disadvantage of full bit vector schemes, is storage overhead. There are two ways to reduce this overhead for a given number of processors while still using full bit vectors. The first is to increase the cache block size; the second is to put multiple processors, rather than just one, in a node that is visible to the directory protocol; that is to use a two-level protocol.
Actually, there are two most interesting alternative solutions: one that reduce the number of bits per directory entry, or *directory width*, and one that reduce the total number of directory entries, or *directory height*, by not having an entry per memory block.

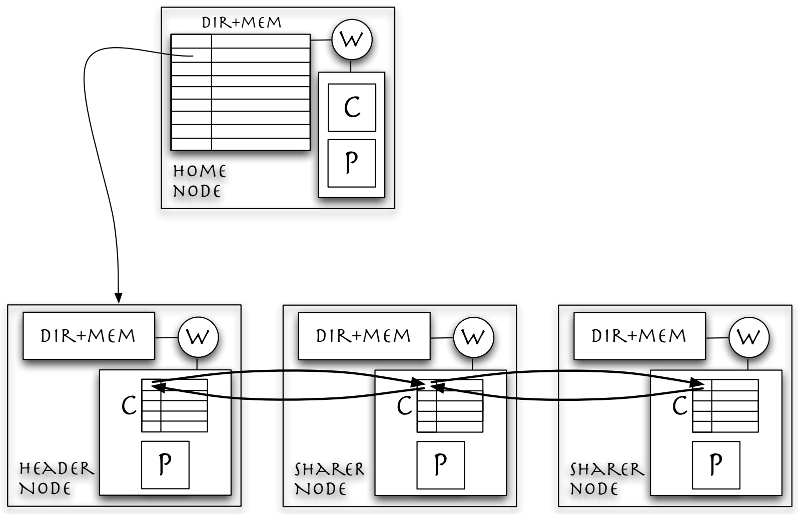


Figure 14: Directory cache-based

Directory width is reduces by using what are called *limited pointer directories*, which are motivated by the observation that most of the time only a few caches have a copy of a block when the block is written. In [1] is presented the idea of maintain a fixed number of pointers, less than the number of the caches in the system. each pointing to a node that currently caches a copy of the block.
Directory height can be reduced by organizing the directory itself as a cache, as presented in [4], taking advantage of the fact that since the total amount of cache in the machine is much smaller than the total amount of memory, only a very small fraction of the memory block will actually be present in caches at a given time, so most of the directory entries will be unused anyway.

Cache-based

In flat, cache-based schemes, there is still a home main memory for the block; however, the directory entry at the home node does not contain the identities of all sharers but only a pointer to the first sharer in the list plus a few state bits. This pointer is called the *head pointer* for the block. The remaining nodes caching that block are joined together in a distribute, doubly *linked list*; that is, a cache contains a copy of the block also contains pointers to the next and previous caches that have a copy, called the *forward* and *backward* pointers, respectively (see figure 14).

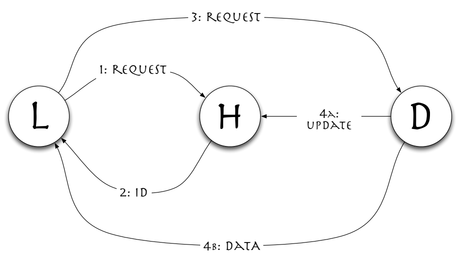
On a read fault:

- the local node sends a request to the home memory to find out the identity of the head node of the linked list, if any, for that block;
- if the head pointer is null (no current sharers), the home replies with with the data;
- if the head pointer is not null, then the requestor must be added to the list of sharers; the home responds to the requestor with the head pointer; the requestor then sends a message to the head node, asking to be inserted at the head of the list and hence to become the new head node; the data for the block is provided by the home if it has the latest copy or by the head node, which always has the latest copy (is the owner) otherwise.

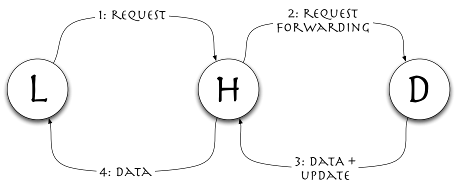
On a write fault:

- the writer again obtains the identity of the head node, if any, from the home;
- it then inserts itself into the list as the head node as before; if the writer was already in the list as a sharer and is now performing an upgrade, it is deleted from its current position in the list and inserted as the new head;
- following this, the rest of the distributed linked list is traversed node by node to fin and invalidate successive copies of the block; acknowledgments for these invalidations are sent to the writer.

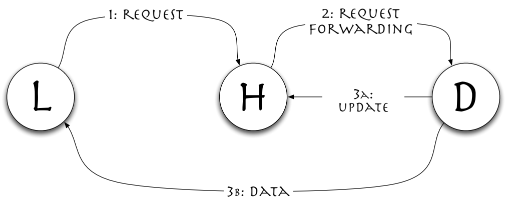
Write backs or other replacements from the cache also require that the node delete itself from the sharing list, which means communicating with the nodes that are before and after it in the list.
Cache-based schemes solve the directory overhead problem found in memory-based schemes. Every block in main memory has only a single head pointer and few state bits. The number of forward and backward pointers is proportional to the number of cache blocks in the machine, which is much smaller then the number of memory blocks. However, manipulating insertion in and deletion from distributed linked lists can lead to complex protocol implementations. These complexity issues have been greatly alleviated by the formalization and publication of a standard for a cache-based directory organization and protocol: the IEEE 1596-1992 Scalable Coherent Interface (SCI) standard, presented in [5].



(a) Strict request-response



(b) Intervention forwarding



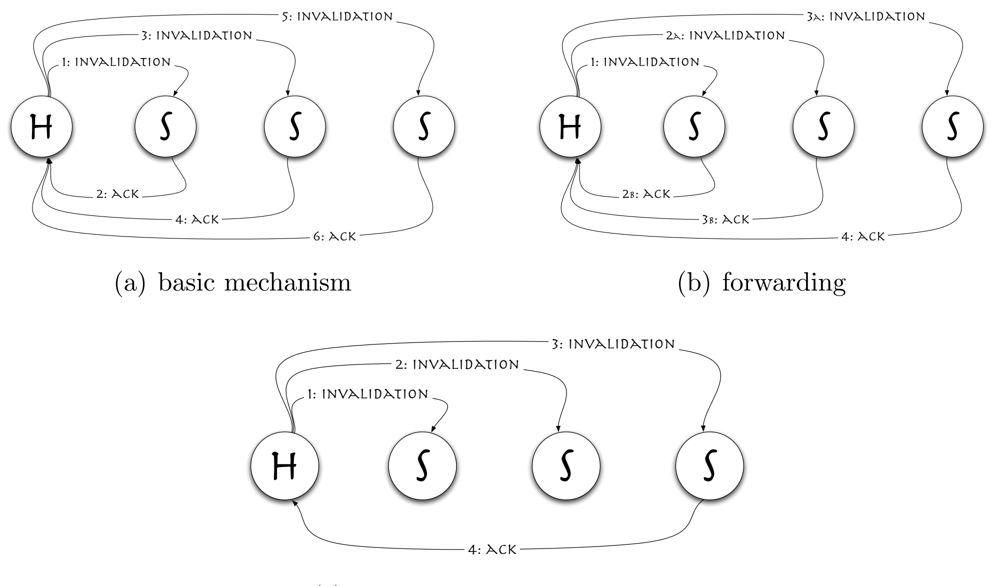
(c) Reply forwarding

Figure 15: Reducing interprocessor communications in a flat, memory-based protocol through forwarding

Flat protocol optimizations

The major performance goal at the protocol level is to reduce the number of interprocessor communications generated by a node as a result of a cache fault.
Consider a read fault to a remotely allocated block that is dirty in a third node in a flat, memory-based protocol. The strict request-response option described earlier is shown in figure 15(a). The home responds to the requestor with a message containing the identity of the owner node. The requestor then sends a request to the owner, which replies to it with the data. The owner also sends a "revision" message to the home, which updates memory with the data and sets the directory state to be shared.

The number of interprocessor communications can be reduced by forwarding of the request from the local node to the dirty node. In particular, the home does not respond to the requestor but simply forwards the request to the owner, asking it to retrieve the block from its cache. As shown in figure 15(b), the owner then replies to the home with the data or an acknowledgment, at which time the home updates its directory state and replies to the requestor with the data.
A more efficient method is *reply forwarding* (figure 15(c)), where when the



(a) basic mechanism

(b) forwarding

(c) forwarding with a single ack

Figure 16: Reducing interprocessor communications in a flat, cache-based protocol

home forwards the request message to the owner node, the message contains the identity of the requestor and the owner replies directly to the requestor itself. The owner also sends a revision message to the home so that the memory and the directory can be updated.
Similar techniques can be used also to reduce the number of interprocessor communications in flat, cache-based schemes when the copies of a block must be invalidated after a write request. In the strict request-response case (see figure 16(a)), every node includes in its acknowledgment the identity of the next sharer on the list, and the home then sends that sharer and invalidation. The total number of transactions in the invalidation sequence is $2s$, where s is the number of the sharers.

Figure 16(b) show how a generic node in the list, when receive an invalidation request, can forwards the invalidation to the next sharer and in parallel sends an acknowledgment to the home.
The number of interprocessor communications can also be reduce if only the last sharer in the list sends a single acknowledgement telling the home that the sequence is done, as shown in figure 16(c).

Correctness of protocols

As we have seen, in directory-based mechanism is not necessary to know the exact status of each copy of a block, in fact, few information is sufficient at the home node, which are a simply main memory's view, which allow the protocol to determine what actions take in order to ensure consistency. Since the nodes communicate through interprocessor communications to execute the actions described by the protocol, the will be periods when a directory's knowledge of a cache state is incorrect since the cache state has been modified but notice of the modification has not reached the directory.
The problem, caused by the distribution of the state, must be resolved; in fact, as a result of a subsequent request for access to the same block, this can handled by the home node using information not yet updated, could affect the accuracy the mechanism adopted for consistency.
Several types of solutions can be used to solve this problem, most of them use additional directory states called *busy states* or *pending states.* A block being in busy state at the directory indicates that a previous request that came to the home for that block is still in progress and has not been completed. When a new request comes to the home and finds the directory state to be busy, coherence may be maintained by one of the following mechanisms.

- *Buffer at the home.* The request may be buffered at the home as a pending request until the previous request that is in progress for the block has completed, regardless of whether the previous request was forwarded to a dirty node or whether a strict request-response protocol was used.

- *Buffer at the local node.* Pending request may be buffered not only at the home but also at the requestors themselves, by constructing a distributed linked list of pending requests. This is a natural extension of a cache-based approach, which already has the support for distributed linked lists. This mechanism is used in the SCI protocol (see [5]).

- *NACK.* An incoming request may be NACKed (refused) by the home, for example by sending a negative acknowledgment to the requestor. The request will be retried later by unit W of the requestor.

- *Forward to the dirty node.* If the directory state is busy because a request has been forwarded to a dirty node, subsequent requests for that block are not buffered at the home or NACKed. Rather, they too are forwarded to the dirty node, which maintains the pending requests. If the block in the dirty node leaves the dirty state before a forwarded request reaches it (for example, due to a write back), then the request

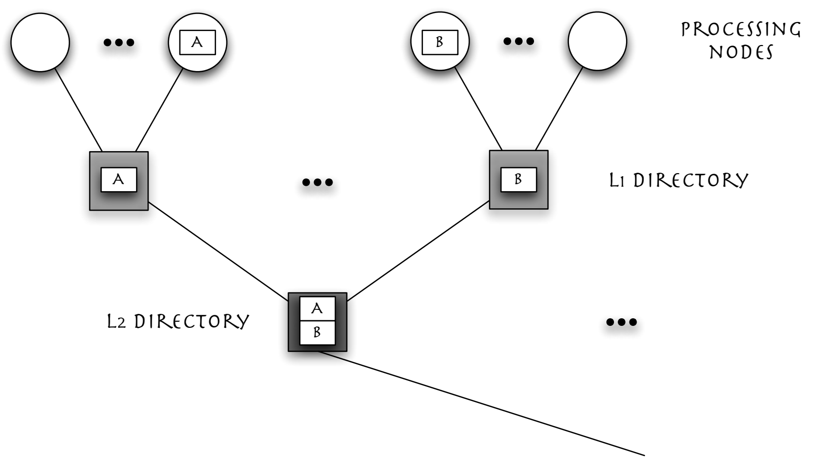


Figure 17: Organization of hierarchical directories

may be NACKed by the dirty node and retried. This approach was used in the Stanford DASH protocol, presented in [7].

These techniques These techniques are not sufficient to ensure that the state transitions of a generic memory block takes place in an atomic way. As explained in section 2.1.1, in snoopy-based cache coherence techniques, atomicity is provided by the bus; whereas in directory-based techniques is not so easy to ensure this property. In fact, when a read exclusive request of a block occurs, it is necessary the home node guarantees an atomic access to the requestor. To do this, it is possible to adopt solutions that temporarily block access requests to the memory module from other nodes, until all the necessary invalidations were not performed.

2.2.2 Hierarchical

In hierarchical schemes, the source of directory information is not known a priori. In fact, the directory information for each block is logically organized as a hierarchical data structure, a tree. The processing nodes, each with its portion of memory, are at the leaves of the tree. The internal nodes of the tree are simply hierarchically maintained directory information for the block: a node keep track of whether each of its children has a copy of a block, as shown in figure 17.

Upon a fault, the directory information for the block is found by traversing up the hierarchy level by level through interprocessor communications until a directory node is reached that indicates its subtree has the block in the

appropriate state. Thus, a node in the system not only serves as a leaf node for the block it contains but also stores directory information as an internal tree node for other blocks.
More detail:

- a read fault flows up the hierarchy either until a directory indicates that its subtree has a copy (clean or dirty) of the memory block being requested or until the request reaches the directory that is the first common ancestor of the requesting node and the home node for that block, and that directory indicates the block is not dirty outside that subtree; the request then flows down the hierarchy to the appropriate processing node to pick up the data; the data reply follows the same path back, updating the directories on its way; if the block was dirty, a copy of the block also finds its way to the home node;

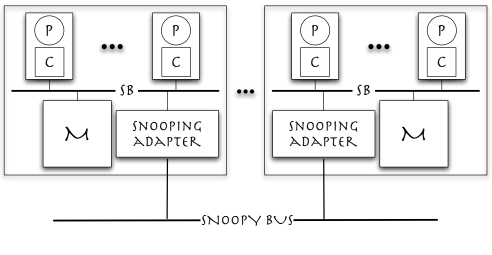
- a write fault flows up the hierarchy until it reaches a directory whose subtree contains the current owner of the requested memory block; the owner is either the home node, if the block is clean, or a dirty cache; the request travels down to the owner to pick up the data, and the requesting node becomes the new owner; if the block was previously in clean state, invalidations are also propagated through the hierarchy to all nodes caching that memory block; finally, all directories involved in the preceding memory operation are updated to reflect the new owner and the invalidated copies.

Hierarchical schemes are not so popular on modern systems due to their performance, in many cases worse compared to the flat schemes.

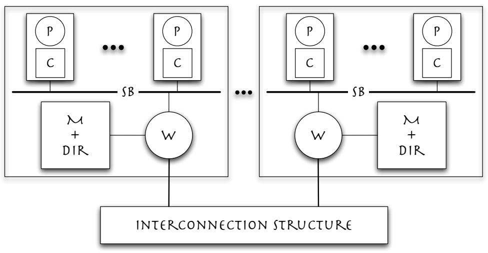
2.3 Hybrid approaches

An alternative solution is to provide a two-level protocol hierarchy. Each node of the machine is itself a multiprocessor. The aches within a node are kept coherent by one coherence protocol called the *inner protocol.* Coherence across nodes is maintained by another , possibly different protocol called the *outer protocol.* Usually, an adapter unit is used to represent a node to the outer protocol.

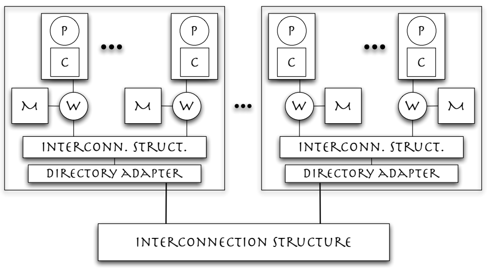
A common organization is for the outer protocol to be a directory protocol and the inner one to be a snooping protocol, as presented in [8] and as shown in figure 18(b). However, other combinations such as snooping-snooping, directory-directory and directory-snooping may be used (see figures 18(a), 18(c) and 18(d) respectively).



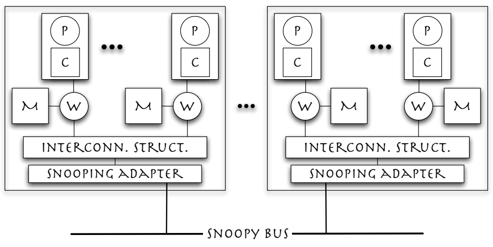
(a) Snooping-snooping



(b) Snooping-directory



(c) Directory-directory



(d) Directory-snooping

Figure 18: Some possible organizations for two-level cache-coherent systems

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 353–362, New York, NY, USA, 1998. ACM.

[2] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[3] Bin feng Qian and Li min Yan. The research of the inclusive cache used in multi-core processor. In *Electronic Packaging Technology High Density Packaging, 2008. ICEPT-HDP 2008. International Conference on*, pages 1 –4, 28-31 2008.

[4] Anoop Gupta, Wolf dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[5] Davib B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, 1992.

[6] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, New York, NY, USA, 2009. ACM.

[7] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM.

[8] Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. *SIGARCH Comput. Archit. News*, 24(2):308–317, 1996.

[9] Edward M. McCreight. The Dragon Computer System: An Early Overview. Technical report, Xerox Corporation, Polo Alto Research Center, Palo Alto, Ca., 94304, December 7, 1984.

[10] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM.

[11] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.