COMPUTER ARCHITECTURE

CONTENTS

29. Cache Coherence I

DR A. P. SHANTHI

The objectives of this module are to discuss about the cache coherence problem in multiprocessors and elaborate on the snoop based cache coherence protocol.

In the previous module, we pointed out the challenges associated with multiprocessors.

The two main challenges that we pointed out are as follows:

1. Parallel and sequential portions of the program

• Our programs are going to have both sequential code and parallel code. As the

sequential code increases, the performance of the multipro Increase Font Size come down. Therefore, we need to write parallel programs that will harness the full power of the underlying parallel architecture.

2. Communication latency

• Communication latency among processors is going to be a major overhead and that has to be reduced. This can be done by caching the data in multiple processors. Caches serve to increase bandwidth and reduce latency of access and are useful for both private data and shared data.

However, when we cache data in multiple processors, we have the problem of cache coherence and consistency. We shall elaborate on that in detail in this module and the next module.

Multiprocessor Cache Coherence: Symmetric shared-memory machines usually support the caching of both shared and private data. *Private data* are used by a single processor, while *shared data* are used by multiple processors essentially providing communication among the processors through reads and writes of the shared data. When a private data is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor. Similarly, when shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces the ***cache coherence*** problem. This is because the shared data can have different values in different caches, and this has to be handled appropriately. Figure 33.1 illustrates the problem. We can see that both processors A and B read location X as 1. Later on, when processor A modifies it to value 0, processor B still has it as value 1. Thus, two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem.*

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Figure 33.1

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called *coherence*, defines what values can be returned by a read. The second aspect, called *consistency*, determines when a written value will be returned by a read.

A memory system is coherent if the following hold good:

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. This ensures that we do not see the older value after the ne wer value.

The first property simply preserves program order, which is true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory. The third property ensures that writes are seen in the proper order.

Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important. We cannot expect that a read of X see the value written for X by some other processor, immediately. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*, which will be discussed in a later module. Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Cache Coherency Protocols: Multiprocessors support the notion of *migration*, where data is migrated to the local cache and *replication*, where the same data is replicated in multiple caches. The cache coherence protocols ensure that there is a coherent view of data, with migration and replication. The key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status:

1. *Directory based*: The sharing status of a block of physical memory is kept in just one location, called the *directory*. The directory can also be distributed to improve scalability. Communication is established using point-to-point requests through the interconnection network.

2. *Snoop based*: Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access. Requires broadcast, since caching information is at processors Useful for small scale machines (most of the market)

We will focus on the snoop based approach in this module. Increase Font Size

Snoopy Cache Coherence Protocol: There are two ways to maintain the coherence requirement. One method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: All other cached copies of the item are invalidated.

The alternative to write invalidate is the *write broadcast* or *write update* mechanism. Here, all the cached copies are updated simultaneously. This requires more bandwidth. Also, when multiple updates happen to the same location, unnecessary updates are done. However, there is lower latency between the write and the read. We shall assume a write invalidate approach for the rest of the discussion.

The bus is normally used to perform invalidates. To perform an invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated. When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes.

Also, we need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. For a write-back cache, the most recent value of a data item can be in a cache rather than in memory. The snooping process is used here also. All processors snoop on the address placed on the bus. If a processor

finds that it has a dirty copy of the requested cache block, it [Increase Font Size] block in response to the read request and causes the memory access to be aborted. In this module, we will examine the implementation of coherence with write-back caches.

The tag bits, the dirty bit and the valid bit that we discussed with respect to caches are used here also. The normal cache tags can be used to implement the process of snooping, the dirty bit to indicate whether the cache block was modified and the valid bit to indicate the validity of the cache block. The only other additional bit that is needed is to indicate whether or not a cache block is shared. For this, we can add an extra state bit associated with each cache block, indicating whether the block is shared. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Every bus transaction must check the cache -address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor.

An Example Snoopy Protocol – MSI: We shall look at an example MSI protocol

and then examine extensions of this basic protocol. In this ca can be in three states – Modified (M), Shared (S) and Invalid (I). Each block of memory is in one state:

- Clean in all caches and up-to-date in memory (Shared)
- Or Dirty in exactly one cache (Exclusive/Modified)
- Or Not in any caches (Invalid)
- Shared : block can be read
- Or Exclusive : cache has only copy, its writeable, and dirty
- Or Invalid : block contains no data

Writes to clean line are treated as misses.

A snooping coherence protocol is usually implemented by incorporating a finite state controller in each node. This controller responds to requests both from the processor and from the bus, changing the state of the selected cache block, as well as using the bus to access data or to invalidate it.

Figure 33.2 shows the requests generated by the processor-cache module in a node (in the top half of the table) as well as those coming from the bus (in the bottom half of the table). The various activities are elaborated below:

1. Read request by the processor which is a hit – the cache block can be in the shared state or modified state – Normal hit operation where the data is read from the local cache.

2. Read request by the processor, which is a miss. This indicates that the cache block can be in any of the following three states:

a. Invalid – It is a normal miss and the read request is placed on the bus. The re-

quested block will be brought from memory and the status will

b. Shared – It is a replacement miss, probably because of an address conflict. The read request is placed on the bus and the requested block will be brought from memory and the status will become shared.

c. Modified – It is a replacement miss, probably because of an address conflict. The read request is placed on the bus, the processor -cache holding it in the modified state writes it back to memory and the requested block will be brought from memory and the status will become shared in both the caches.

3. Write request by the processor which is a hit – the cache block can be in the shared state or modified state.

a. Modified – Normal hit operation where the data is written in the local cache.

b. Shared – It is a coherence action. The status of the block has to be changed to modified and it is hence called upgrade or ownership misses. Invalidates will have to be sent on the bus to invalidate all the other copies in the shared state.

4. Write request by the processor, which is a miss. This indicates that the cache block can be in any of the following three states:

a. Invalid – It is a normal miss and the write request is placed on the bus. The requested block will be brought from memory and the status will become modified.

b. Shared – It is a replacement miss, probably because of an address conflict. The write request is placed on the bus and the requested block will be brought from memory and the status will become modified. The other shared copies will be invalidated.

c. Modified – It is a replacement miss, probably because of an address conflict. The write request is placed on the bus, the processor-cache holding it in the modified state writes it back to memory, is invalidated and the requested block will be brought from memory and the status will become modified in the writing cache .

5. From the bus side, a read miss could be put out, and the cache block can be in the shared state or modified state

a. Shared – Either one of the caches holding the data in the shared state or the memory will respond to the miss by sending the block

b. Modified – A coherence action has to take place. The block has to be supplied to the requesting cache and the status of the block in both the caches is shared.

6. The bus sends out an invalidate when a write request comes for a shared block. The shared block has to be invalidated and this is a coherence action.

7. From the bus side, a write miss could be put out, and the cache block can be in the shared state or modified state

a. Shared – It is a write request for a shared block. So, the block has to be invalidated and it is a coherence action.

b. Modified – A coherence action has to take place. The block has to be written back and its status has to be invalidated in the original cache .

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Figure 33.2

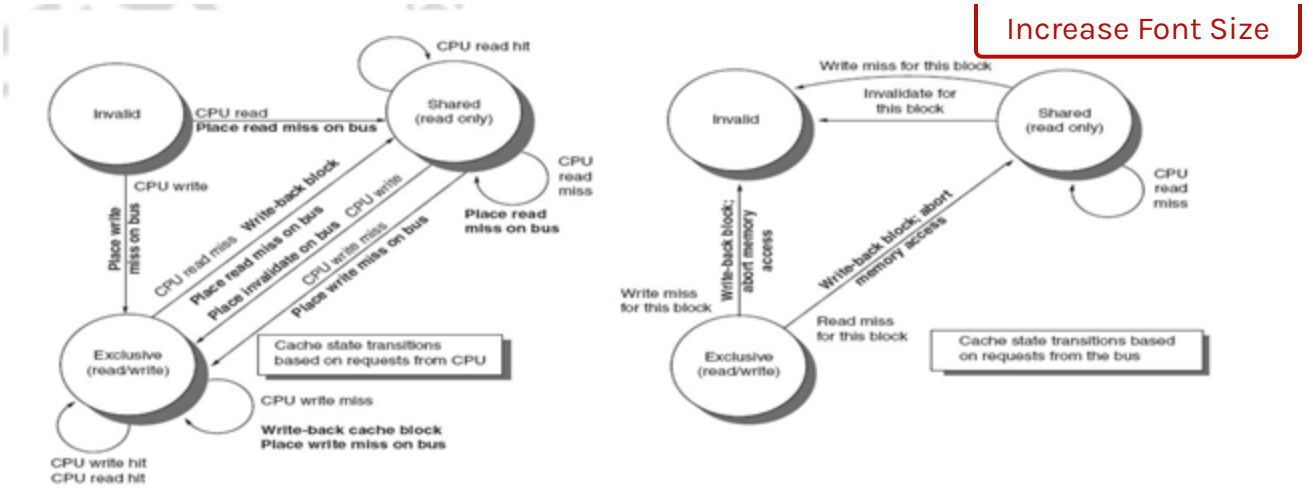

Figure 33.3

Figure 33.3 shows a finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure 33.2), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom half of the table in Figure 33.2). The state in each node represents the state of the selected cache block specified by the processor or bus request. Figure 33.3 provides a combined view.

Increase Font Size

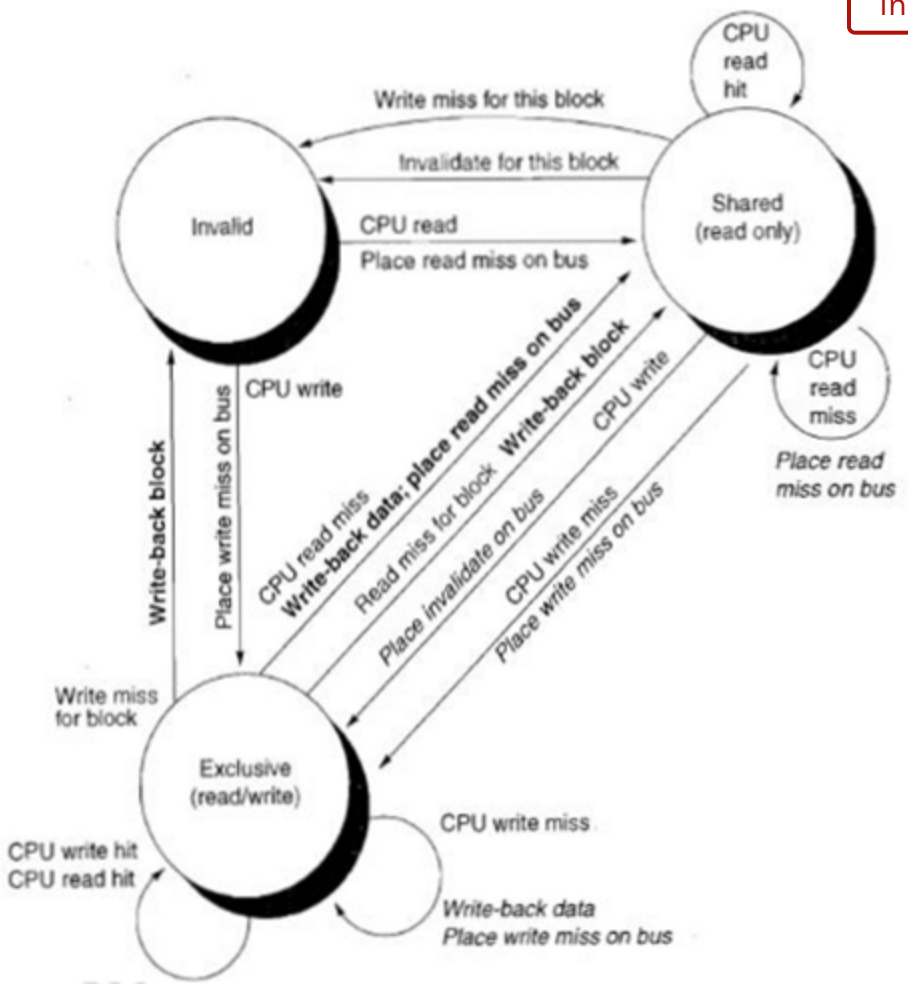

Figure 33.4

Implementation Complications: Though the protocol discussion seems to be simple, there are a number of complications in the implementatio n. The foremost problem is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality this is not true. Similarly, if we used a switch, as all recent multiprocessors do, then even read misses would also not be atomic. Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue.

Limitations in Symmetric Shared-Memory Multiprocessor Protocols: As the number of processors in a multiprocessor grows, or as the memory demands of each processor grow, any centralized resource in the system can become a bottleneck. In the simple case of a bus -based multiprocessor, the bus and the memory become a bottleneck. So, scalability becomes an issue. In order to increase the communication bandwidth between processors and memory, designers have used multiple buses as well as interconnection networks, such as crossbars or small point-to-point networks. In such designs, the memory system can be configured into multiple physical banks, so as to boost the effective memory bandwidth while retaining uniform access time to memory.

Extensions to the MSI Protocol: The basic MSI protocol is extended by adding other states in order to provide optimizations.

1. The first extension is adding an exclusive state, the **MESI** (Modified, Exclusive, Shared and Invalid) protocol. The exclusive state is added to indicate a clean block in only one cache. This prevents the need to sending write invalidates on a write, since the block is available only in one cache. The processor merely changes the state to modified. However, when a read miss to the block in the E state occurs, the status has to changed from exclusive to shared, in order to maintain coherence.

2.The other extension that Intel i7 uses is adding an additional state to the MESI protocol, called the **Forwarding** state, leading to the **MESIF** protocol. This identifies a sharing cache that will forward the data block, when there is a request. When a cache block is shared by multiple caches, and there is a yet another read miss, the memory can provide the block or one of the caches can provide the data. The cache block, which is designated as the forwarding block, is responsible for forwarding the data. This avoids contention among the various caches in providing data. Also, to make sure the forwarding cache does not replace this block, the forwarding status is granted to the cache that has most recently acquired this block.

3. Yet another extension is the addition of an **Owned** state indicating that the cache is the owner of the block and is not up to date in memory. This is the

MOESI protocol. Normally, when a block in the modified state Increase Font Size tus of both the blocks will be made shared and the block will be written back to memory. But, in the MOESI protocol, the status of the original cache is changed from modified to owned, indicating that the memory does not have a copy and only this cache has the updated copy. It is also the responsibility of this cache to supply the data on a miss. It will write it back to memory when the block is replaced. The AMD Opteron uses the MOESI protocol.

To summarize, we have defined the cache coherence problem in multiprocessors. We have defined the two types of cache coherence protocols. The snoop based cache coherence protocol has been discussed in detail. The implementation issues, the limitations and the extensions to the basic protocol have been discussed.

Web Links / Supporting Materials

- Computer Architecture – A Quantitative Approach , John L. Hennessy and David A.Patterson, 5th Edition, Morgan Kaufmann, Elsevier, 2011.