## Lecture #14a: Cache Coherence Directory Based Protocols

Course: Computer Architecture III

Lecturer: Miodrag Bolic

Scribe: Michel Bray and Marc Branchaud

Date: Friday, October 27th 2006

### Introduction

The topic of this lecture is directory-based protocols for cache coherence. Cache coherence problems occur in multiprocessor systems. Cache coherence refers to the integrity of data stored in local caches of a shared resource. [1] Whenever a processor equipped with data cache accesses a variable, the value of this variable is copied into the processor’s cache. The coherence problem occurs when two or more processors access the same data element and at least one processor modifies its value. If no cache coherence protocols are implemented different processors are not made aware of the modifications made to the data and different processors keep using an invalid value of the variable, this in turn means that the final results of the program will be invalid.

Two general categories of cache coherence protocols exist; they are “Snoopy protocols” and “Directory protocols”. Snooping is the process where the individual caches monitor address lines for accesses to memory locations that they have cached. When a write operation is observed to a location of which a cache has a copy, the cache controller invalidates its own copy of the snooped memory location. [1] A directory-based protocol uses hard-ware to keep track of shared variables. When shared variables are accessed and/or modified by different processors, different bits are set or reset in the directory to indicate whether the cached data is valid or not.

### Explanation

**2 What is a directory based protocol?**

A directory based protocol is a hardware based scheme used to eliminate cache coherency problems. A list of every processor that holds the same instances of certain blocks of data is kept in a directory which in turn is kept in shared memory. An entry of a block in a directory will hold pointers to each cache that holds a copy of the data and a dirty bit that permits or denies a cache to update the block. [2]

**3 When to use a directory based protocol?**

When snoopy protocols are used in networks, the broadcasting techniques used in these protocols can become costly. For example: when a snoopy protocol is used in a 2D mesh, the update information is sent to every processor in the mesh creating high traffic in the interconnection network, hindering efficiency. [3]

**2d mesh**

Figure 1: Broadcast signal through a 2D mesh. [4]

On the other hand, directory based protocols send information to specific processors listed in the block entry of the directory. This protocol discourages the use of the entire interconnection network in a multi-processor environment. In our previous example of the 2D mesh, the update information is sent only to the processors that hold a copy of the block, which stops the use of the complete interconnection network when it is not needed. [2]

**2d mesh no broadcast**

Figure 2: A directory based protocol working through a 2D mesh. The green circles contain a copy of the same variable. [4]

**4 Types of directory protocols**

Directory-based cache coherence protocols can be classified in one of two general categories; they are either “centralized” or “distributed”; [2]both of these have the common goal of reducing traffic on the interconnection network. In order to achieve this goal both categories can support multiple shared copies of a memory block. [2]

**4.1. *Centralized Directory Protocols***

As the name implies these protocols use a single directory to track shared memory content. This category of protocol can be implemented in more than one way.

**4.1.1 *Full-map Protocol***

This protocol implements a directory containing information on every cache entry. In such an implementation of the centralized directory protocol every directory entry contains “presence bits” and a “single-inconsistent bit”.

full-map

Figure 3**:** Full map protocol directory [2]

The directory contains one presence bit associated with every available cache. When a presence bit is set to ‘1’ it means that the associated cache contains a copy of the data block associated with the directory entry. In the case that one and only presence bit is set to ‘1’ then the single-inconsistent bit can be set to ‘1’ this means that only the associated processor has writing privileges to that memory element.

*Read miss*

In the event that the single-inconsistent bit is set, the memory sends an update request to the cache with the private bit set. The private cache then sends the information to the requesting cache, clears its own private bit and the single-inconsistent bit is cleared in the block entry. [2]

Whether or not the single-inconsistent bit is set, the requesting cache is sent a copy of the block and the present bit is set for the cache. Once the cache receives the copy, the valid bit is set and the private bit is cleared. [2]

*Write miss*

When a write miss is occurs, all other caches are invalidated and their present bits are cleared. The invalidated caches will then send acknowledgements to the memory. During the acknowledgements, if one of the newly invalidated caches happens to have its private bit set, then the memory block is updated with that cache’s data. Once all acknowledgements are received, the requesting cache’s present bit is set as well as the single-inconsistent bit. Finally, the cache receives a copy of the information, modifies it and the valid and private bits are set. [2]

*Write hit*

In the case where the private bit is clear, the requesting cache sends a privacy request to the memory. Other caches are invalidated similar to the write miss. The single-inconsistent bit is set and an acknowledgement is sent from the memory to the requesting cache. The cache’s private bit is then set.

**4.1.2 *Limited Directories***

The limited directory protocol solves the potential directory size problem by setting a fixed directory size. This means that there are a fixed number of pointers in the directory regardless of the number of processors. This limits the number of copies found in multiple caches and therefore reduces the size of the directory and the search time. [3]

**limited-map**

Figure 4: Limited directory (based on figure 3 [2])

Limited directories are similar to full-map directories in the way that they handle a read miss, a write miss and a write hit. The only difference is that when a cache requests a copy of a block but there is no room for a new pointer in the entry, then a previous pointer is replaced by the new pointer. When this happens, the block in the old cache is set to invalid. The pointer to be replaced is chosen by a replacement policy to the designer’s discretion. [2]

**4.2 *Distributed Directory Protocols***

The distributed directory protocols have the same functions as the centralized directory protocols. However the distributed directory protocols differ from their centralized counterparts in the way the directory is implemented. These protocols partition and spread the directory among the caches and/or memories. This helps reduce the directory sizes and memory bottlenecks in large multiprocessor systems. [2]

**4.2.1 *Hierarchical Directory Protocols***

This type of directory protocol is mostly used in systems comprised of a set of connected clusters. Each cluster contains a set of processing units and a directory connected by an interconnection network. A request that cannot be serviced by the caches within a cluster is sent to the other clusters as determined by the directory. [2] This type of architecture is like a group of centralized directories.

**4.2.1.1 *Centralized Directory Invalidate***

This protocol is one possible implementation of a distributed directory protocol. More specifically it uses a hierarchical implementation of the directory.

When this protocol is used and a write-request is issued within a cluster, the cluster’s centralized directory establishes which processors have a copy of the memory block to which the processor wishes to write. Once the directory has established which processor has a copy of the data block, invalidating signals and a pointer to the requesting processor are forwarded to all processors that have a copy of the block. [3] The invalidating signal ensures that only the writing processor has a valid copy of the memory block.

*Write-miss*

When a write miss request is received the directory establishes which processors have a copy of the memory block and proceeds to send the appropriate processors an invalidating signal as well as a pointer to the requesting processor. Once the invalidating signal is received by the processors they invalidate themselves and send an acknowledgement signal to the requesting cache. When the invalidating process is complete only the requesting cache will have read-write access to the memory block.

Cache C0

X:

X:

Directory

Cache C1

Cache C2

Cache C3

Data

X:

Data

Data

1

0

1

0

Memory

Write

inv-ack

inv-ack

Invalidate &

requester

Invalidate &

requester

Write-reply

Figure 5: A write operation performed by processor 3 [3]

**4.2.2 *Chained directory protocols***

Chained directories are similar to full-map directories since there is an unlimited amount of caches that can hold a copy of the block entry. To solve the directory size problem that is present in full-map directories, a chained directory makes use of a singly or doubly linked list meaning the directory holds a single pointer to a cache. That cache then holds a pointer to the next cache until a cache is found with a terminator pointer (CT). [3] The directory itself will only hold one pointer which is the head of the list, the first cache with the copy of the information.

X:

X:

Directory

C0

Data

X:

CT

Data

C2

Data

Cache C0

Cache C1

Cache C2

Cache C3

Memory

Figure 6: Chained Directory [3]

**4.2.2.1 *Scalable Coherent Interface (SCI)***

Scalable coherent interface protocols are doubly linked lists that fall under the categorization of chained directory. Every block address found in the directory; whether it is in shared memory or in a cache, contains additional tag bits. Tag bits in the memory contain a pointer to the first cache with the block, also called the head. The tag bits in the cache contain the next and previous list entries. [3]

*Read miss*

Before the block is cached, meaning there is no head pointer in the directory, the block in memory is in the uncached state and cached copies are invalid. The cache requesting the memory is sent a copy of the information. The directory is put into cached state and the head pointer in the directory points to the cache requesting the information. [2]

If the block is already cached and a new cache requests the block, the head pointer is set to the new cache requesting the block. The backwards pointer on the new head cache is set to the block in memory. A request is then sent to the old head cache, which sets its backwards pointer to the new head cache and sends its data to the cache.[3]

Memory

Cache C0

(head)

Cache C2

(Invalid)

1) read

2) prepend

**Before**

Memory

Cache C0

(middle)

Cache C2

(head)

**After**

Figure 7: Sharing list addition [3]

*Write miss*

When a cache requests a write-miss to the memory, the head pointer in the directory is set to the requesting cache. The old head cache sets its forward pointer to the new head cache. The new head cache sets its backwards pointer to the memory block. The new head cache then proceeds to send an invalidate request through its forward pointer, invalidating the next cache. That cache, once invalidated, sends an invalidated request to the next cache and so on until every cache except the head cache is invalidated. Only once every cache is invalidated does the new head cache write the information. [2]

*Write hit*

If the cache requesting the write is the only cache in the linked list, then the write will proceed immediately. [2]

When the cache is already the head cache, every other cache becomes invalidated. After the invalidation, the head cache is written. This is similar to the write miss algorithm. [2]

When the cache is already the head cache, every other cache becomes invalidated. After the invalidation, the head cache is written. This is similar to the write miss algorithm. [2]

If the cache is not the head cache, then the cache will be detached from the linked list to place itself next to the current head cache. The cache requesting the write becomes the new head cache. Similarly to the write miss and the previous write hit, the new head cache will invalidate all the other caches before writing the new information.

In the case of every successful write, the size of the linked list will always be reset to 1.

Cache C0

(tail)

Cache C2

(middle)

Cache C3

(head)

Memory

Purge

Purge

Figure 8: Head purging other entries [3]

**4.2.2.2 *Stanford Distributed Directory (SDD)***

This protocol is similar to the SCI protocol however this one is based on a singly linked list implementation of a chained directory. As with the SCI protocol memory points to the head of the sharing list. However since this protocol is based on a singly linked list each processor can only point to the previous processor in the chain and cannot point to the next processor. When the sharing list must be extended or shortened these functions are handled differently than from the SCI protocol.

*Read-miss*

When a read-miss is encountered by a processor, a read-miss signal is sent to the memory. Upon receiving this signal the memory proceeds to updating its head pointers to contain the address of the requesting processor. After having updated the head pointer, the memory then sends a read-miss-forward signal to the old head. When the old head receives this message it sends the requested information to the new head along with its own address. [3] The new head then proceeds to copy the data and uses the address to aim its pointer towards the old head of the sharing list.

Memory

Cache C0

(middle)

Cache C2

(head)

**After**

Memory

Cache C0

(head)

Cache C2

(Invalid)

1) read

3) read-miss-reply

**Before**

2) read- miss-

forward

Figure 9: Sharing list addition [3]

*Write-miss*

In the event of a write-miss request, the requesting processor sends a write-miss message to the memory. Upon receiving this message the memory proceeds aiming its pointer to the requesting processor. A write-miss-forward message is also sent to the old head. When such a message is received the old head sends a write-miss-reply-data message to the requesting processor. This processor also proceeds to invalidating itself and forwards the write-miss-forward message to the next processor on the list who repeats the process of invalidating itself and forwards the same message to the next processor in the list. When the last processor of the list receives the message is proceeds to invalidating itself and sends a write-miss-reply message to the requesting processor. When the requesting processor has received both the write-miss-reply-data and the write-miss-reply the write-miss process is completed. [3]

Cache C3

(exclusive)

Memory

**After**

Cache C0

(tail)

Cache C2

(head)

Cache C3

(invalid)

Memory

1) write

3) write- miss-forward

2) write- miss-forward

3) write- miss-reply-data

4) write- miss-reply

**Before**

Figure 10: Write miss sharing list removal [3]

### Summary

The cache coherence problem is of great importance when dealing with shared memory multiprocessing systems. Fortunately as discussed in this document many solutions were developed in order to solve the coherency problem. These solutions are valid regardless of the architecture of the system.

The full map directory protocol would be the easiest one to implement however it is very costly in terms of memory size. Furthermore, when using a full map directory we run the risk of having a large amount of unused resources since not every processor will be sharing every variable therefore large portions of the directory would simply be set to ‘0’. This implementation also has the disadvantage of not being very scalable since every time a new processor is added to the system the directory size has to be increased.

The limited directory protocol helps reduce the cost of the memory. This protocol achieves this by having a limited amount of data slots for directory entries. This protocol also helps with the problem of scalability when using this implementation. If a new processor is added to the system the directory does not require any modifications. This is possible since processes are already in place to deal with missing data slots in the directory. Furthermore each slot is not associated with a single processor but rather contains the address of the processor in question.

Distributed directory protocols further reduce the cost of memory and increase the scalability of the system. These protocols partition and spread the directory among the available caches and memories. When using these protocols the directory contains the addresses of the processors.

Each directory protocol has its own way of dealing with read and write operations. Unlike their snooping counterparts the directory-based protocols do not rely on broadcasting signals to function thus reducing traffic on the interconnection network and therefore increasing the system’s efficiency. [3]

References

1. Wikipedia, “Cache Coherence”, 2006, available at <http://en.wikipedia.org/wiki/Cache_coherence>, (last accessed October 30, 2006)
2. M. R. Zargham, *Computer Architecture Single and Parallel Systems*. Upper Saddle River, New Jersey: Prentice-Hall, Inc,1996, pp.252-257
3. H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architercture and Parallel Processing*. Hoboken, NJ: John Wiley & Sons, 2005, pp.89-96
4. M. Bolic. CEG4131. Class Lecture, Topic: “Cache Coherence.” CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Oct 27, 2006.