Computer Architecture III Study Guide Multiprocessor Systems

Maximillien Courchesne-Mackie (mcour010@uottawa.ca)

I. Shared Memory Systems

A. Uniform Memory Access (UMA)

All processors in the UMA model shared the physical memory uniformly. The latency involved in any processor accessing the shared memory is independent of the requesting processor.

B. Non-Uniform Memory Access (NUMA)

When arranged in a NUMA model, memory access time varies with the location of the word in memory. The shared memory is distributed across all nodes in the form of local memories. The local memories form a global address space which can be accessed by any processor.

C. Cache-Only Memory Architecture (COMA)

The difference between COMA and NUMA is that in COMA architecture the local memories are used as caches instead of main memories. In this architecture, there is no home node for data.

D. Cache Coherent NUMA (ccNUMA)

In NUMA architecture, maintaining coherent caches across all nodes brings with it a lot of overhead. In ccNUMA, cache controllers can communicate with each other to keep a consistent map of which cache stores which data.

II. Message Passing Systems

Message passing systems do not share memory and, consequentially, are much more scalable. In these systems, messages can be passed synchronously or asynchronously between threads. Seeing as each thread has its own private memory, SEND and RECV commands are used to pass data in between threads.

A. Synchronous Message Passing

In a synchronous message passing system, a SEND command will block and wait for the corresponding RECV signal to be received. Similarly, a RECV will block and wait for the corresponding SEND command before continuing execution. These two rules guarantee that each thread has the proper data. However, synchronous message passing is prone to deadlocks (for example, two SEND commands sent at the same time between two threads will cause them to block indefinitely). Another downside is that this message passing scheme pairs synchronization and communication into one primitive, which can cause performance issues.

B. Asynchronous Message Passing

In an asynchronous message passing system, when one thread issues a SEND command, it continues to execute code without waiting for the RECV signal. There are two types of asynchronous message passing: *blocking* and *non-blocking*.

A blocking SEND will not give control back to the issuing thread until the data has been successfully sent. Likewise, a blocking RECV will wait for the data to be copied into local memory before giving control back to the thread.

A non-blocking primitive will give control back to the issuing thread immediately after sending the signal so that other code can be executed.

III. Cache Coherence

A system is said to be cache coherent if all processors see each datum as it was last globally written.

A. Snoopy Protocol

Snoopy cache coherence protocols gets their name from the fact that all caches connected to the bus can ”snoop” every message and service those that concern them. In this kind of system, depending on the protocol, on a write hit, the modified value and its address are added to the request buffer. When granted, the request is placed on the bus and each processor can ”snoop” to see if they have a local copy of the modified data. If found, remote copies of the data will be marked as *invalid* (V=0). Finally, the local cache is updated with the modified value and the bus is released to service another cache controller.

There are, however, design issues regarding a snoopy cache coherence protocol. These lie mainly with the fact that the signals sent out by each node in response to a snoop keeps the whole bus occupied until the full transaction is completed. To fix this, we decouple requests and responses so that the bus can be open to other processes. This is called a *split-transaction bus*. These types of buses have higher bandwidth, but also higher latency due to an added step: arbitration. The arbitrator must match requests with responses. One way of doing this is by assigning a unique ID to each request which will be added to the response when it is placed on the bus. This way the response is properly sent to the node which requested it.

Another issue with a split-transaction bus is that the FIFO buffers in between the different levels of cache, the memory banks and the bus could potentially fill up and cause a buffer overflow. This can be mitigated by adding an acknowledgement phase to sending a request or a response. If the ACK comes back negative, the request or response is tried

again until the buffers have free space. With more levels of private caches (and, in extension, more sets of FIFO buffers in between them), however, it takes longer to respond to a request. This latency can be alleviated with cache inclusion. Cache inclusion means that whatever blocks exist in a higher cache level, must also exist in the same node's lower cache levels. With this in action, operations involving caches can be handled at the lowest level of cache (nearest to the bus) without traversing the cache system.

1) Transient States: With a split-transaction bus, the cache controller can no longer decide what the next state of the cached block is, seeing as something could happen to change the next obvious state before a response is received on the bus. To mitigate this, caches need to have *transient states* for cached blocks. A block will be in a transient state so long as the transaction is being completed and will revert to a stable state once a response has been received.

B. MSI Protocol

The MSI cache coherence protocol dictates that each block in cache be in one of three states: modified (M), shared (S) or invalid (I). The modified state indicates that the data in cache is different from the data in main memory and must be written back upon eviction. A shared state indicates that the data exists in at least one cache and can be evicted without writing it back to main memory. Finally, an invalid state indicates that the data is invalid (has been changed elsewhere) and must be re-fetched from memory or another cache.

When a read is requested and the block in question is in the M or S state, the cache supplies the data. If it is marked as invalid (I), the cache checks other caches to see if the data exists in the M state. If it does, the remote cache must write the data back to main memory, the local cache must copy the data from main memory into itself and the data in both caches must change their states to shared (S).

When a write is requested, if the data is in the modified state, it is written to local cache. If it is in the shared state, the cache controller must notify other caches with local copies of the data that they must evict it. The block is then modified locally. If another cache has a modified version of the block, it must write it back to main memory and the local cache must read it before being modified.

C. MOESI Protocol

The MOESI protocol contains the same three states introduced in the MSI protocol, but adds two more: owned (O) and exclusive (E). The owned status means that the block is valid and that other copies exist in other caches, however only that cache has exclusive rights to make changes to it. This allows a block marked as modified to be shared across caches without having to be written back to main memory. The exclusive status means that the cache is the only existing copy of the block and it is unmodified.

D. Directory Protocol

A directory-based cache coherence protocol has a directory (physical entity) which keeps track of each block and which cache is currently storing a copy of it. It wastes less bandwidth than a snoopy protocol because the request doesn't have to occupy every node on the bus.

Directories can be centralized or distributed. A central directory keeps information about all blocks in one central data structure. This is not ideal as searching through the directory could cause bottlenecks due to its large size. Decentralized directories allows each memory module to maintain a separate directory.

1) Full Map Directory: Full map directories contain N pointers, where N is the number of processors. For each memory block, an N-bit vector is kept. Each bit in this vector corresponds to one processor. If the bit is high, then that processor's cache contains a copy of the block. This method doesn't scale very well due to the vector needed one bit per processor (many processors in a system means a much larger directory).

In the event of a read miss, the directory will do one of two things. If the requested block doesn't appear in any other caches, it will service the requesting cache a copy of the block from main memory. If the block does exist in another cache (there is at least one high bit in the directory's vector), the directory tells the other caches to send the block back to memory for an update and to send a copy to the requesting cache. The requesting processor can then read the block from its local cache and the directory vector is updated.

When a write miss is triggered, the directory sends an invalidate signal to the sharing caches (which send an ACK signal back to memory). The directory vector is then updated and the writing processor has exclusive rights to the block.

2) Limited Directory: A limited directory has a fixed number of pointers per entry regardless of the amount of processors in the system. This restricts the number of block copies which can be shared between caches and solves the full map directory's size problem, however, the number of shared copies that can exist in the system is limited.

3) Chained Directory: A chained directory is similar to a full map directory but the directory itself is distributed among the caches. They keep track of shared block copies by maintaining a chain of directory pointers. The main directory (which sits near the main memory) only keeps a pointer to one cache node. In turn, that cache node keeps a pointer to another node which contains a copy of the block and so on.