CIS 629: Project No.5 DASH Q and A

Sameer S. Shende

December 2, 1994

1. **Cache Coherence Problem.**

   Coherence in caches used in multiprocessor system ensures that with shared data, the most recent value of a variable will be read by a processor irrespective of the location of the cache which contains the data. To illustrate the need for cache coherence consider the case in where there are two processors A and B in a multiprocessor machine. If X is a shared variable accessed by programs running on these CPUs concurrently, then both can access the variable X. The processors A and B have caches to improve the performance of the system. Consider a symmetric multiprocessor system with global memory and one shared common address space.

   Let processors A and B access X (read operation). As a result of this X is present in both the caches of A and B. It is marked as a shared data in the main memory. The main memory has the latest value of X at this stage. Suppose processor A writes to X, then the copy of X in the local cache of A is updated. Let the new value of X be denoted as X'. Suppose, there was no cache coherence mechanism to indicate the change in X to B and main memory, then we can consider after an interval of time, processor B requesting for the contents of X. At this time, the cache would give the contents of the data X held in its memory which would be different from X'. This results in a wrong value being processed by B which could result in errors.

   In a system which implements cache coherence, the changed value of X (X') would be seen by the processor B any time after A completes the write operation on variable X. This can be achieved by a write update or a write invalidate scheme where the copy of X in B stores the new value (X') or is tagged as holding a dirty value respectively. Thus, all processors in a multiprocessor system would get access to the latest value of X. Cache coherence does not guarantee that the value of the shared data will not change by the time the value gets to the processor B.

2. **Directory based cache coherence.**

   Cache coherence can be implemented using snoopy bus protocols or directory based cache coherence protocols. In a snoopy bus protocol, the updated value of a shared chached datum is broadcast over a bus to the other caches to update (this is write update scheme) or an invalidate request is broadcast so that other processors which have the datum can mark the datum as invalid. A directory based cache coherence protocol on the other hand does not braodcast the value to other processors but does a multicast operation (more of a point to point scheme). A directory based protocol can be implemented as a Central directory or as a Distributed directory scheme. This directory contains information about the processor which "owns" the datum (in the case of write) and the value of the datum.

- Central Directory scheme
  A central directory scheme duplicates all cache directories in a multiprocessor with individual caches. So, when a refernce is directed to the central directory it is associatively searched to get access to the data. The drawback of using this scheme is that the search operation takes a considerable amount of time and there is scope for contention when a number of processor request for data and the directory cannot cope up with the requests.
- Distributed Directory scheme
  In a distributed directory scheme, the directory is duplicated on every Memory module (typically shared memory in a cluster). So, the number of directories may be less than the number of caches and processors. In this scheme, a directory does not duplicate the entries (one address cannot be associated with more than one directory). In this case also the directory maintains information about the caches which contain the addresses which are shared. When a read request comes for a shared data item, the requestors cache gets the data and the item is marked with this processor too (in case other processors were also sharing it). When a write request coms, the other caches which contain copies of the data are sent either an update or invalidate request.
  The directories can be implemented as Full Map, Limited or Chained.
- Full Map Directories
  In a full map implementation, the directory entry is implemented with a dirty bit (which can be set or cleared) and one bit per processor. The bit is set or clear which represents whether the data is present in a particular processors cache or not. In this case the bit position represents the processor id which corresponds to it. The dirty bit is set when a processor requests a write operation. In this case only the bit of the processor which "owns" the data is set and the rest of the bits are set. Thus, the cache entry is tagged as valid/invalid and the data is writable or not writable. The disadvantage of a full map directory scheme is that as the number of processors grows the number of bits per memory location have to increase and so it is not scalable with the number of processors.
- Limited Directories
  The problem of scalability of the above mentioned scheme is solved partially by Limited Directories (or partial map directory scheme). In this case the shared memory contains a dirty bit (valid/invalid state) and a fixed number of pointers (to processor caches) along with the block of data. In this case if the number of pointers per data item is m, then a maximum of m processors (¡ total number of processors) can access the data. The processors (whose cache contains the shared data item) are pointed to by the pointers in the directory entry. When a processor requests a write operation on one of them, it sets the bit as dirty and sends update or invalidate requests to the other caches which hold the data. When the number of processors exceeds the limit (m), the directory evicts cached copies. The disadvantage of this scheme is that it must store pointers (or processor ids which take logN storage - for each of the N processors) and so it scales NlogN.

- Chained Directories
  In this scheme the restrictions on the number of cached copies is removed as the directory contains only one pointer per block of data (with a bit). This pointer points to the data item in the cache (the first item). In this case, the cache is different in the sense that each data field also has a pointer to the next data item which is shared. This way, when n processors share a data item (in each processors cache), the directory has the pointer to the first cache item which points to the next and so on. This way an entire "chain" of the directory entry is maintained. The invalidations (or updates) are sent along the chain till the end on a write operation. The chain of directory pointers is doubly linked to handle cache replacements (in case an item is thrown out of the cache by the cache replacement policy like the LRU). This scheme scales as NlogN (N is the processor count).

3. **DASH protocol for a remote read to a shared data.**

   Dash implements two levels of cache (write thorugh on the first level with four write buffers followed by a write back cache) and a shared cluster memory with a Remote access cache(henceforth denoted as RAC) and a directory controller (henceforth denoted as DC) per cluster. The physical memory is arranged according to the address such that the address determines the cluster where it can be found. A block of data in the directory can be in three states viz., uncached-remote (or not cached by any remote cluster), shared-remote (or cached in an unmodified state by one or more remote clusters) or dirty-remote(cached in a modified state in a single remote cluster).

   On a remote read to the shared data, the cache sends the request to the bus and on determining that it is a remote address, the RAC checks to see that it does not have the data and the DC sends a request to the cluster (home) where th e data is situated (from the address). When the request is dispatched, the RAC allocates space for the reply (so if another request for the same data occurs from the same cluster, a second message is not sent). At the home cluster the Pseudo CPU (henceforth denoted as PCPU) which polls for remote requests picks up the request and issues it on the home cluster's bus. The home cluster directory then looks up its entry on the block of data requested. If the block of data is not cached in the home cluster(uncached-remote) or in a shared-remote state, the block of data is sent to the requesting cluster and the PCPU of the requesting cluster gets it and it is sent out on the bus and the CPUs in the cluster which requested for it get the data and it is simultaneously stored in the RAC.

   If the block of the data in the home cluster happens to be in a dirty-remote state, then that means that some other remote cache holds the data and the so the home directory redirects the request for the data to the cluster which has the data. After the owner cluster finds the data (again using the PCPU), it sends out two messages. One message which contains the block of data is sent out to the requesting cluster and the PCPU takes care of it as explained above. The second message is sent to the home cluster and the block of data is copied into the memory and the state of the block is changed in the home cluster from dirty-remote to shared-remote. The owner cluster now tags the block as read only. When two requests for a block of data are encountered by the home cluster (obviously from different clusters) and the block is marked as dirty-remote, then both the requests are forwarded to the owner cluster(thus making the home clusters PCPU a "state-less" server). When the owner cluster sends the two messages after the first request it marks its block as read only. So, the second request does not find a dirty block and results in a NAK - negative acknowledgement being sent to the second requesting cluster which retries the request and sends it again to the home cluster.

The diadvantage in this scheme is that one can always come up with a scheme where a write request interspersed with two remote requests always results in the second request being retried again and again and not getting the data (its theoretically possible but difficult to illustrate in real life).

4. **DASH protocol for a remote write to a shared data.**

   One of the ways to deal with latency is relaxed consistency (among Coherent caches, relaxed consistency, prefetching and multiple contexts). DASH uses release consistency model. With this, two fence operations are implemented. They are, full-fence (which blocks the processor and completes all the reads and writes which are pending) and write-fence which delays write operations while allowing the process to continue.

   As in the case of read to a shared data illustrated above, the requesting cluster sends the write request to the home cluster which sends invalidate requests to the clusters that were sharing the data block (if the block is marked as shared-remote or dirty-remote) and marks the block as dirty-remote andgives ownership of the block to the requesting cluster (if it is shared-remote then it also gives the no. of invalidates) . The invalidate requests are processed by the remote clusters and an Ack is sent by each cluster to the requesting cluster. At the requesting cluster, it receives the ownership of the block and the number of acks it should receive to know that the write operation has completed. Now, the processor receives the ownership and continues with its operation. Each invalidate ack is received by the clusters reply controller (RC) which keeps track of each processors pending invalidates and decrements this count each time it receives an Ack. The remote access cache (RAC) keeps track of the number of entries per processor and sends the corresponding invalidate Acks to the RC. When a write-fence is requested (wait till all read writes complete) by a processor, it blocks till all the acks corresponding to the processor are received and the counter in the RC for it goes to zero.

5. **SGI PowerChallenge Similarities with DASH**

   SGI Powerchallenge was a shared memory SMP system with two levels of cache hierarchy. The first level cache consisted of a 32KB on-chip instruction cache and 16 KB data cache and the second level was a 2MB DRAM (upto 4MB) per CPU. The cache coherence scheme used in the second level cache of both DASH and SGI is the Illinois protocol which is essentially a write invalidate scheme. However, The DASH similarity ends here. The dash has a cluster based distributed directory protocol which cannot be compared to the protocol in SGI.

   The SGI protocol uses cache to cache transfers only for dirty data and each element in the cache has four states invalid, exclusive, dirty and shared. The DASH states are based on the cluster - shared-remote, uncached-remote, dirty-remote. The similarity is in the write-invalidate scheme which is used by both. If we were to look at the cache of only one cluster in DASH then we can find many similarities but not if compare the distributed directory based scheme of DASH to SGI PowerChallenge.