CMP SCI 635: Modern Computer Architecture

The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor

Name: Kunjal Panchal Date: 14th Nov, 2019 Student ID: 32126469

Paper: 19 – Parallelism [Lenoski90]

Strengths:

1. To efficiently interleave long duration remote transactions with the short duration local transactions; they extended the MPBUS protocol to support a retry mechanism. Remote requests are signalled to retry while the inter-cluster messages are being processed. To avoid unnecessary retries the processor is masked from arbitration until the response from the remote request has been received. When the response arrives, the requesting processor is unmasked, retries the request on the bus, and is supplied the remote data.

2. The processor is able to overlap the fetching of the data with useful work. When the processor is ready to use the prefetched data, it issues a normal read or read exclusive request. These operations cause the directory controller to send out a read or read-exclusive request for the data, but do not block the processor. The latency for the data will be reduced.

Weaknesses:

1. Release Consistency requires performing write propagation in bulk at the release point of synchronization. Propagating such a large number of writes altogether will slow down the release access and the subsequent acquire access. Hence it can hardly improve the performance of a hardware cache coherence system.

2. In wormhole routing, contiguous flits in a packet are always contained in the same or adjacent nodes of the network. This can cause difficulties, as possibility of deadlock arises. Deadlock in the interconnection network occurs when no message can advance towards its destination because the queues of the message system are full. No communication can occur over the deadlocked channels until exceptional action is taken to break the deadlock.

Questions/Assertions:

1. Since the cost of populating a cache is almost always higher than an uncached request, with the cost amortized over subsequent uses of the cached object, unnecessary cache invalidations tend to hurt performance. That gets worse with shared caches, as a cache invalidation usually involves either implicit serialization or synchronization of pending accesses by other consumers.

2. CPUs have memory fences. Modern CPUs execute out-of-order, but L1, L2, and L3 caches also innately change the order of which memory operations happen. Case in point: one-hundred memory reads will become one memory read from DDR4 Main Memory, and then 100-memory reads to L1 cache.

   But if another core changes the memory location, how will the CPU Core learn about it? Memory fences (aka: flushes) can forcibly flush the cache, write transaction buffers, and so forth to ensure that a memory operation happens in the order the programmer expects.

3. Do we care about locality of descriptors between threads in the shared cache? If not, then aligning structures on cache line size shouldn't be uncommon?

4. Misconceptions about caches often lead to false assertions, especially when it comes to concurrency and race conditions. For example, the common refrain that concurrent programming is hard because "different cores can have different/stale values in their individual caches". Or that the reason we need volatiles in languages like Java, is to "prevent shared-data from being cached locally", and force them to be "read/written all the way to main memory".

   In the case of volatiles, the solution is pretty simple – force all reads/writes to volatile-variables to bypass the local registers, and immediately trigger cache reads/writes instead.

5. Hardware caches on modern x86 CPUs like Intel's, are not strongly consistent. Intel CPUs support Total Store Ordering (TSO) which still allows the following to print 00:

   ```
   thread 1  | thread 2
   ---------+------------
   a = 1;       | b = 1;
   print(b);  | print(a);
   ```