McCalpinJohn
Black Belt

11-15-2016 04:21 PM
94 Views

I usually implement a

I usually implement a producer/consumer code using "data" and "flag" in separate cache lines. This enables the consumer to spin on the flag while the producer updates the data. When the data is ready, the producer writes the flag variable. At a low level, the steps are:

1. The producer executes a store instruction, which misses in its L1 Data Cache and L2 cache, thus generating an RFO transaction on the ring.
    1. The data for the store is held in a store buffer at least until the producer core has write permission on the cache line in its own L1 Data Cache.
    2. The data for the store may be held in the store buffer for longer periods, awaiting other potential stores to the same cache line.
2. The RFO traverses the ring to find the L3 slice that owns the physical address being used, and the RFO hits in the L3.
3. The L3 has the data, but also sees that it is marked as being in a clean state in one or more processor private caches, so it issues an invalidate on the cache line containing the flag variable.
    1. The L3 may or may not have a way of tracking whether the cache line containing the flag variable is shared in another chip.
    2. The L3 may have directories that track which cores may have a copy of the cache line containing the flag variable. If there are directories, the invalidates may be targeted at the specific caches that may hold the cache line, rather than being broadcast to all cores.
    3. The L3 may send the data for the cache line containing the flag to the producer's cache at this time, but that data cannot be used until the coherence transactions are complete.
4. The consumer receives the invalidate, invalidates its copy of the flag line, and responds to the L3 that the line has been invalidated.
    1. Immediately after responding to the L3 that the line has been invalidated, the spin loop in the consumer tries to load the flag variable again.
5. The L3 receives the invalidation acknowledgements from all the cores that may have had a shared copy of the line, and notifies the producer core that it now has write permission on the cache line containing the flag data.
    1. If you are lucky, the producer core will write the flag variable from the store buffer to the L1 Data Cache immediately on receipt of permission.
    2. If you are unlucky, the producing core may lose the line containing the flag variable before the store buffer is written to the L1 Data Cache. I don't know if Intel processors can do this, but I know some processors than can lose the line before dumping the store buffer.

6. Very shortly after sending the write permission notification to the producer core, the L3 will receive a read request from the consumer core for the same cache line containing the flag.
    1. Depending on the implementation, several different things might happen.
    2. One option is for the L3 to hold the line containing the flag variable in a "transient" state while it waits for an acknowledgement from the Producer core that it has received the write permission message. In this case the L3 will either:
        1. Stall the read request from the consumer core, or
        2. NACK the read request from the consumer core (i.e., tell it to try again).
    3. Another option is for the L3 to immediately process the read request and send an intervention/downgrade request for the cache line containing the flag variable to the producer core's cache.
7. In the "lucky" case, the intervention/downgrade request generated by the read from the consumer core will get the new value of the cache line containing the flag variable and return it to the consumer core and to the L3 slice that owns that physical address.
    1. Various implementations have specific ordering requirements here that determine whether the cache line must be sent to the L3 first, then the to consumer core, or whether it can be sent to both at the same time.
    2. Some implementations require an extra handshaking step after the consumer core receives the data, before the L3 will give it permission to use the data. (This is more common in the case of a store than a read.)
8. Finally the consumer core gets the new value of the flag variable and sees that it has changed! The data is now ready!
9. The spin loop on the consumer core now exits, which incurs a 20-cycle mispredicted branch delay.
10. The consumer core now executes a load instruction to get the data. This misses in the consumer's L1 and L2 caches and generates a read request on the ring.
11. The read request traverses the ring to the slice that owns the physical address of the cache line containing the data (which may be a different slice than the one controlling the cache line containing the flag), and the read request hits in the L3.
12. The data in the L3 is stale, but the L3 knows exactly which core has write permission on the cache line containing the data, so it issues an intervention/downgrade on the cache line containing the data and targeting the cache of the producer core.
13. The cache(s) of the producer core receive the intervention/downgrade request and return the new value of the cache line containing the data variable to the L3, simultaneously downgrading the permissions on the line so that it is now "read-only" in the producer's caches.
14. As was the case for the cache line containing the flag variable, the cache line containing the data variable makes its way to both the L3 slice that owns the physical address and the consumer core that requested the data.
15. The cache line containing the data arrives in the consumer core's cache, and the load instruction is allowed to complete.
16. Once the consumer core has gotten the data safely into a register, it typically has to re-write the flag variable to let the producer core know that the value has been consumed and that the producer core is free to write to the cache line containing the data variable again.

1. This requires the consumer to make an "upgrade" request on the cache line containing the flag, so it can write to it. This is similar to the sequence above, but since the consumer already has the data, it does not need the L3 to send it again -- it only needs to wait until all other cores have acknowledge the invalidate before it can write to the flag line.
2. Double-buffering can be used to avoid this extra transaction -- if the consumer uses a different set of addresses to send data back to the producer, then the fact that the producer has received another message from the consumer means that the consumer must have finished using the original data buffer, so it is safe for the producer to use it again.

There are many variants and many details that can be non-intuitive in an actual implementation. These often involve extra round trips required to ensure ordering in ugly corner cases. A common example is maintaining global ordering across stores that are handled by different coherence controllers. This can be different L3 slices (and/or Home Agents) in a single package, or the more difficult case of stores that alternate across independent packages.

There are fewer steps in the case where the "data" and "flag" are in the same cache line, but extra care needs to be taken in that case because it is easier for the polling activity of the consumer to take the cache line away from the producer before it has finished doing the updates to the data part of the cache line. This can result in more performance variability and reduced total performance, especially in cases with multiplier producers and multiple consumers (with locks, etc.).

"Dr. Bandwidth"

0 Kudos

Share Reply