A Primer on Memory Consistency and Cache Coherence (Chapter 3-4)

by Yunqi Zhang
04/15/2013

1. Memory Consistency Model (MC Model)
    a. give rules that partition executions into those obeying MC and those disobeying MC
    b. compare to coherence:
        i. coherence: per cache block basis
        ii. consistency: across all blocks
2. Sequential Consistency (SC)
    a. memory order respects each core's program order, where memory order is the order of operations
    b. memory order respects each core's program order
    c. naive implementations
        i. uniprocessor: executing all threads on a single sequential core
        ii. switch: switch selects to execute one operation from one of the multi-cores in its original program order
        iii. (+) allow SC executions
        iv. (-) does not scale up with increasing core count, sequential performance bottleneck
    d. basic implementation with cache coherence
        i. Single-Writer-Multiple-Reader invariant cache coherence
        ii. each core has its own cache
        iii. (+) operations can be done concurrently in parallel if they do not conflict
    e. impacts from more features
        i. non-binding prefetching: As the loads and stores are performed in program order, it doesn't matter in what order coherence permissions are obtained
        ii. speculative cores: similar to non-binding prefetching that no effect on SC
        iii. **A core can issue coherence requests in any order (don't have to be program order)**
        iv. dynamically scheduling: check required
            1. speculatively accessed block has not left the cache
            2. replay each speculative load check if the value loaded is equal to previous one
        v. non-binding prefetching in dynamically scheduling:
            1. **SC only requires a core's loads and stores access its level-one cache in program order**
            2. **SC dictates the order in which loads and stores get applied to coherent memory but does NOT dictate the order of coherence activity**

        vi. multithreading: need to ensure a thread cannot read a value written by another thread on the same core before the store has been made VISIBLE

    f. atomic operations
        i. atomically perform "read-modify-write": "test-and-set, fetch-and-increment"
        ii. spin-lock: use RMW to atomically read whether lock's value is unlocked
    g. MIPS R10000
        i. issue loads and stores in program order into address queue

3. Total Store Order
    a. motivation: hide the latency of frequently happened stores
    b. single-core: bypassing the value of the most recent store to A to succeeding loads
    c. TSO
        i. preserve the program order for: 1) load->load, 2) load->store, 3) store->load
        ii. omit store->load (does not matter for most programs)
        iii. programmers can prevent non-SC execution by adding FENCE for above
    d. loads and stores leave each core in that core's program order <p
    e. a load either bypasses a value from the write buffer or awaits the switch as before
    f. a store enters the tail of the FIFO write buffer or stalls the core if the buffer if full
    g. when the switch selects core Ci, it performs either the next load or the store at the head of the write buffer
    h. **A thread my bypass values that it has written, but other threads may not see the value until the store is inserted into the memory order**
    i. FENCE: specify that all instructions before the FENCE in program order must be ordered before any instructions after the FENCE in program order