Module 15: "Memory Consistency Models"
Lecture 34: "Sequential Consistency and Relaxed Models"

Memory Consistency Models

- Memory consistency
- SC
- SC in MIPS R10000
- Relaxed models
- Total store ordering
- PC and PSO
- TSO, PC, PSO
- Weak ordering (WO)

[From Chapters 9 and 11 of Culler, Singh, Gupta]
[Additional reading: Adve and Gharachorloo , WRL Tech Report, 1995]

Previous Next

Module 15: "Memory Consistency Models"
Lecture 34: "Sequential Consistency and Relaxed Models"

Memory consistency

- Coherence protocol is not enough to completely specify the output(s) of a parallel program
  - Coherence protocol only provides the foundation to reason about legal outcome of accesses to the same memory location
  - Consistency model tells us the possible outcomes arising from legal ordering of accesses to all memory locations
  - A shared memory machine advertises the supported consistency model; it is a "contract" with the writers of parallel software and the writers of parallelizing compilers
  - Implementing memory consistency model is really a hardware-software tradeoff: a strict sequential model (SC) offers execution that is intuitive, but may suffer in terms of performance; relaxed models (RC) make program reasoning difficult, but may offer better performance

SC

- Recall that an execution is SC if the memory operations form a valid total order i.e. it is an interleaving of the partial program orders
  - Sufficient conditions require that a new memory operation cannot issue until the previous one is completed
  - This is too restrictive and essentially disallows compiler as well as hardware any re-ordering of instructions
  - No microprocessor that supports SC implements sufficient conditions
  - Instead, all out-of-order execution is allowed, and a proper recovery mechanism is implemented in case of a memory order violation
  - Let's discuss the MIPS R10000 implementation

SC in MIPS R10000

- Issues instructions out of program order, but commits in order
  - The problem is with speculatively executed loads: a load may execute and use a value long before it finally commits
  - In the meantime, some other processor may modify that value through a store and the store may commit (i.e. become globally visible) before the load commits: may violate SC (why?)
  - How do you detect such a violation?
  - How do you recover and guarantee an SC execution?
  - Any special consideration for prefetches?
    - Binding and non-binding prefetches
- In MIPS R10000 a store remains at the head of the active list until it is completed in cache
  - Can we just remove it as soon as it issues and let the other instructions commit (the store can complete from store buffer at a later point)? How far can we go and still guarantee SC?
- The Stanford DASH multiprocessor, on receiving a read reply that is already invalidated, forces the processor to retry that load
  - Why can't it use the value in the cache line and then discard the line?
- Does the cache controller need to take any special action when a line is replaced from the cache?

Relaxed models

- Implementing SC requires complex hardware
  - Is there an example that clearly shows the disaster of not implementing all these?
    - Observe that cache coherence protocol is orthogonal
  - But such violations are rare
  - Does it make sense to invest so much time (for verification) and hardware (associative lookup logic in load queue)?
  - Many processors today relax the consistency model to get rid of complex hardware and achieve some extra performance at the cost of making program reasoning complex
  - P0: A=1; B=1; flag=1; P1: while (!flag); print A; print B;
  - SC is too restrictive; relaxing it does not always violate programmers' intuition
- Three attributes
  - System specification: which orders are preserved and which are not; if all program orders are not preserved what support is provided (software and hardware) to enforce a particular order that the programmer wishes
  - Programmer's interface: set of rules, if followed, will lead to an execution as expected by the programmer; normally specified in terms of high-level language annotations and labels
  - Translation mechanism: how to translate programmer's annotations to hardware actions
- Let's take a look at a few relaxed models: TSO, PSO, PC, WO/WC, RC, DC

Total store ordering

- Allows a read to bypass (i.e. commit before) an earlier incomplete write
  - This essentially means a blocked store at the head of the ROB can be removed (but remains in write buffer) and subsequent instructions are allowed to commit bypassing the blocked store
  - Can hide latency of write operations
  - Note that this is the only allowed re-ordering
  - Programmer's intuition is preserved in most cases, but not always
  - P0: A=1; flag=1; P1: while (!flag); print A; [same as SC]
  - P0: A=1; B=1; P1: print B; print A; [same as SC]
  - P0: A=1; print B; P1: B=1; print A; [violates SC]
  - Implemented in many Sun UltraSPARC microprocessors
- How do I enforce SC in the last example if I really care?
  - May be needed when porting this program from R10000 to UltraSPARC
  - Must ensure that a read cannot bypass earlier writes
  - Microprocessors provide "fence" instructions for this purpose
  - SPARC v9 specification provides MEMBAR (memory barrier) instruction of different flavors
  - Here we only need to use one of these flavors, namely, write-to-read fence just before the load instruction
  - This fence will not allow graduation of load until all stores before it graduates
  - If fence instruction is not available, substituting the read by a read-modify-write (e.g., ldstub in SPARC) also works

PC and PSO

- Processor consistency (PC) is same as TSO, but does not guarantee write atomicity
  - The writes may become visible to different processors in different order
  - P0: A=1; P1: while (!A); B=1; P2: while (!B); print A;
  - BTW, how can a system not guarantee write atomicity? What hardware logics you get rid of by not implementing write atomicity?
  - Implemented in Intel processors
- Partial store ordering (PSO) allows reads as well as writes to bypass writes
  - Still implements write atomicity like TSO
  - Further overlaps write latency
  - Deviates a lot from SC (flag spinning no longer works)
  - Store barrier instruction is needed to “emulate” SC

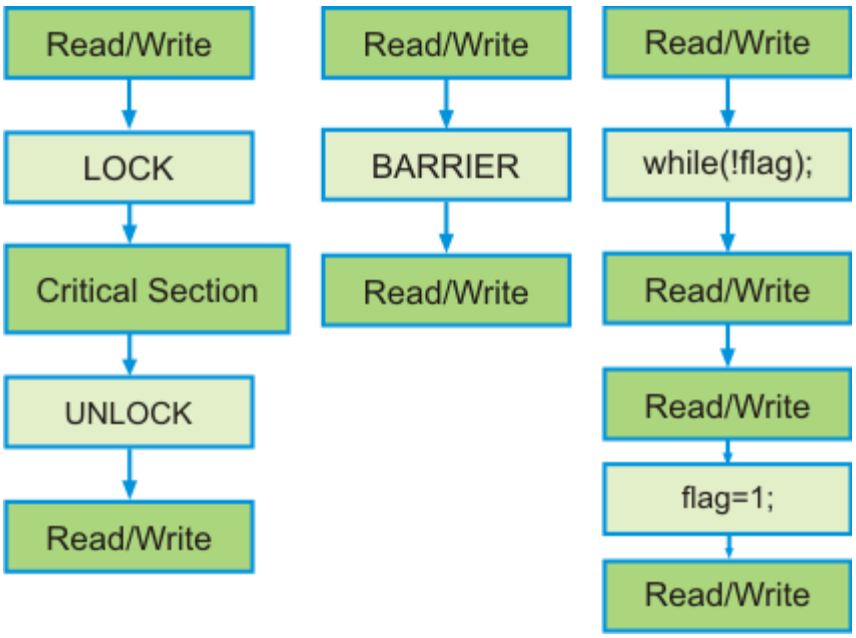
TSO, PC, PSO

- Hardware support
  - TSO and PC still need to worry about load-invalidate squash (why?)
  - In TSO and PC the store queue entries are freed in-order because stores are not allowed to bypass stores
  - A processor supporting PSO can retire stores in any order (however, a store cannot bypass a load)
  - The store barrier instruction (stbar in SPARC) goes through the normal store queue and controls issuing of future store instructions
  - Note that in all three models memory operations to the same or overlapping addresses must issue in program order
  - All three models may benefit from bigger write/store buffers

Weak ordering (WO)

- First seminal relaxed consistency model; also known as weak consistency
  - An important observation is that a parallel programmer does not really care about ordering among memory operations between synchronizations as long as conventional data and control dependencies are preserved within a process
  - For example, within a critical section the exact order in which independent reads/writes are executed is not important
  - Weak ordering makes use of this property and relaxes all memory orders between synchronization operations e.g., within and outside critical sections, between consecutive barriers, before and after flag synchronization
- A few constraints must be followed
  - Any code from outside a critical section cannot be re-ordered with codes inside the critical section i.e. all codes before a critical section must commit before lock is acquired, all codes within a critical section must commit before releasing the lock, and any code after a critical section must not issue before releasing the lock
  - Any code before a barrier must commit before entering a barrier and any code after a barrier must not issue before leaving the barrier (second condition should hold naturally if barrier implementation itself is same)
  - Any code before flag wait must commit before waiting on the flag, any code after flag

wait must not issue before the flag is set by producer, any code before setting of a flag must commit before setting the flag, and any code after setting of flag must not issue before setting of flag



- Perfectly suits modern microprocessors and aggressive compiler optimizations
  - Either hardware must be able to recognize synchronization operations and stall until everything before it has graduated or compiler must insert proper memory barrier instructions
  - MIPS R10000 provides a sync instruction and a fence count register for this purpose; fence count register is incremented whenever an L2 miss leaves the chip and decremented on a reply; a sync instruction disables issuing from address queue until fence register is zero and all outstanding memory operations have committed
  - A processor supporting WO does not need to have load-invalidate squash as in the MIPS R10000

Previous Next