Memory Consistency Models

(Architectural Support for Improved Understanding of Program Outcome)

Mainak Chaudhuri
mainakc@cse.iitk.ac.in

Agenda

- What are these models and why
- Sequential consistency model
- Relaxed consistency models
  - Total store order

What are these models and why?

- A parallel program can have multiple possible outputs

  T0: x=1; T1: y=x; print y;

  What does T1 print if the initial value of x is 0?

- Coherence protocol is not enough to completely specify the output(s) of a parallel program
  - Coherence protocol only provides the foundation to reason about the legal outcomes of accesses to the same memory location
  - Memory consistency models tell us the possible outcomes arising from legal ordering of accesses to all memory locations

What are these models and why?

- Without any specified ordering constraints, all possible outputs could be legal

  T0: A=1; print B; T1: B=1; print A;

  What do T0 and T1 print if the initial values of A and B are zero?

- Given a memory consistency model, only a subset of outcomes is possible and it is important for a programmer to know this subset
  - A shared memory machine advertises the supported memory consistency model; it is a "contract" with the writers of parallel applications and system software

What are these models and why?

- Usually, the programmer wants one specific output from a correct program
  - Suppose the programmer wants T1 to print 1 in the following program

  T0: x=1; T1: y=x; print y;

  - It may seem that synchronization can guarantee such an output (flag=0 initially)

  T0: x=1; flag=1; T1: while(!flag); y=x; print y;

  - What if flag=1 is made visible to other processes before x=1 becomes visible?
    - Either by compiler re-ordering or hardware re-ordering (latter must still honor precise exception)
  - This is quite possible given that the two stores are going to two different addresses

Memory consistency models

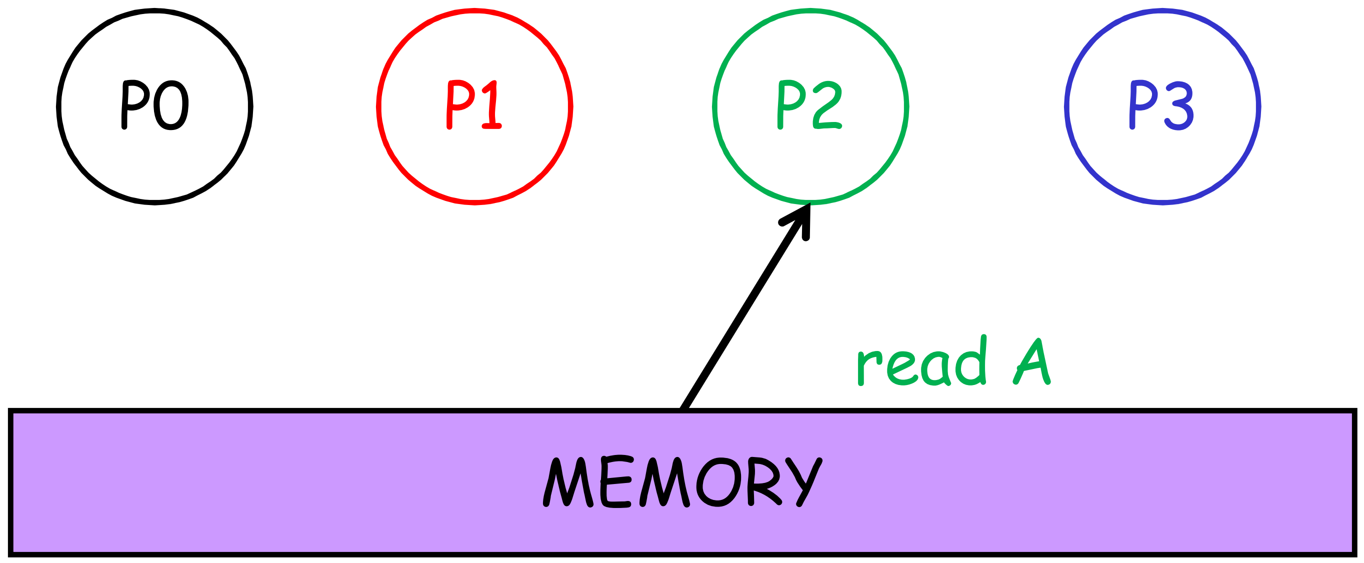
- A memory consistency model is a set of rules that specify the set of allowed orderings between all memory accesses
- A multiprocessor normally advertises the supported memory consistency model
  - This essentially tells the programmer what the possible correct outcome(s) of a program could be when run on that machine
  - A multiprocessor is said to implement a memory consistency model when it adheres to the set of ordering rules specified by that model

Sequential consistency

- Many memory consistency models exist
  - Each model represents a unique point in the three-dimensional space spanned by ease of programming, implementation complexity, and performance/energy
  - Sequential consistency (SC) is the most intuitive one and we will focus on it first
- Legal SC orders
  - Achieved by interleaving accesses from different processors
  - The accesses from the same processor are presented to the memory system in program order

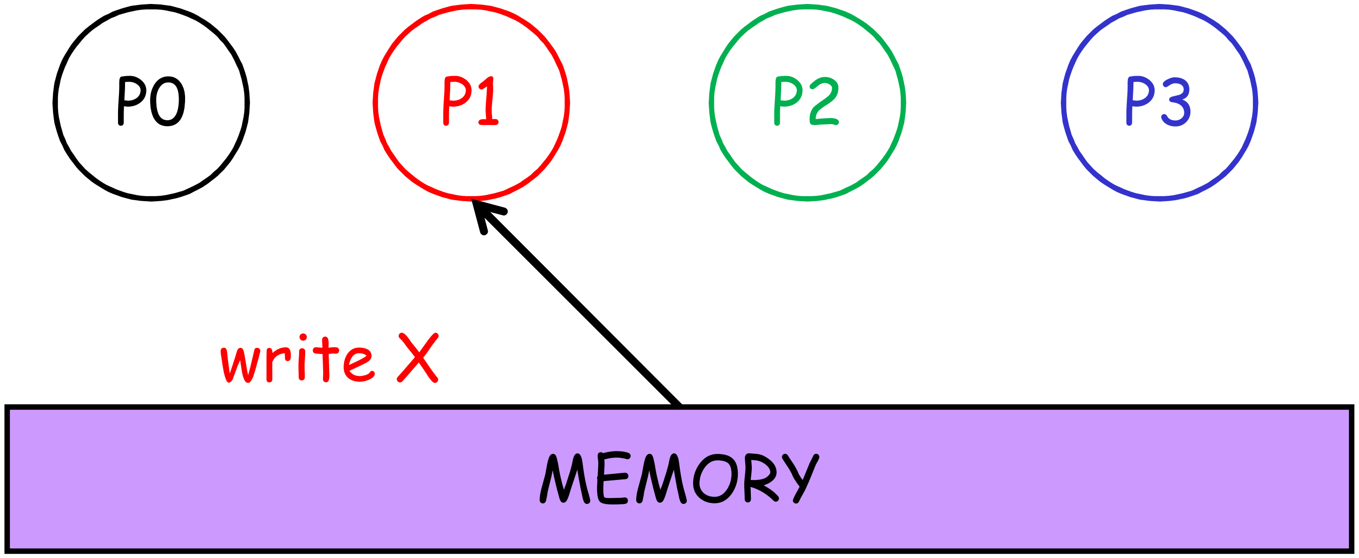
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
  - Picks the next access from a randomly chosen processor



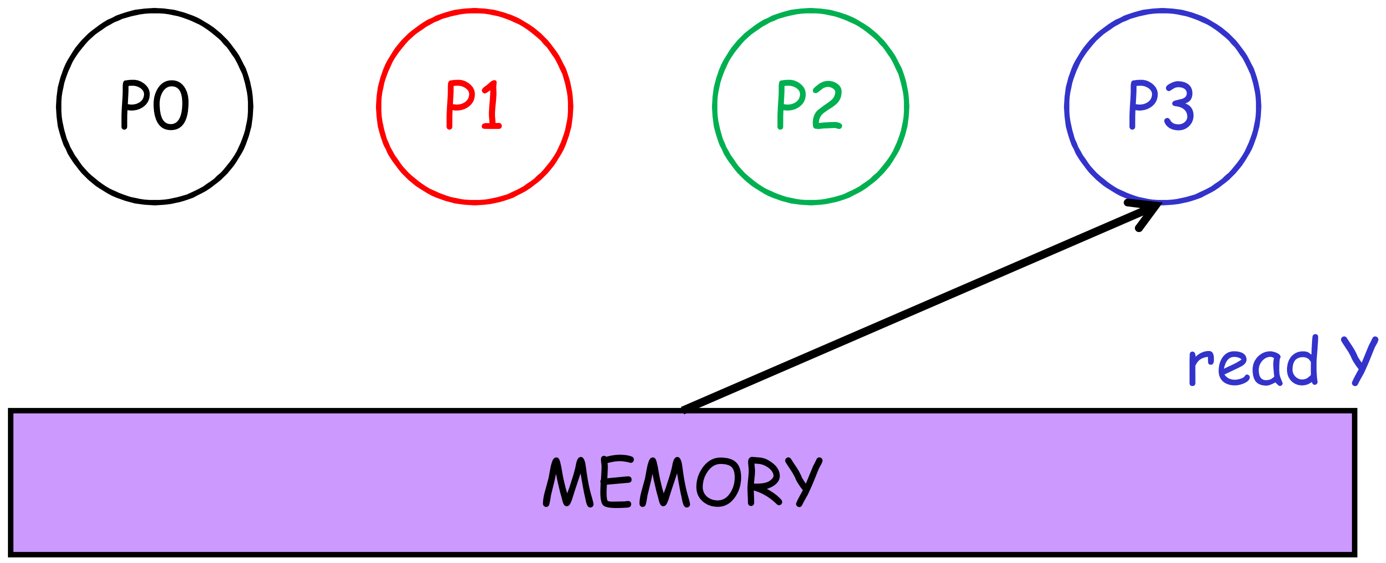
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
  - Picks the next access from a randomly chosen processor



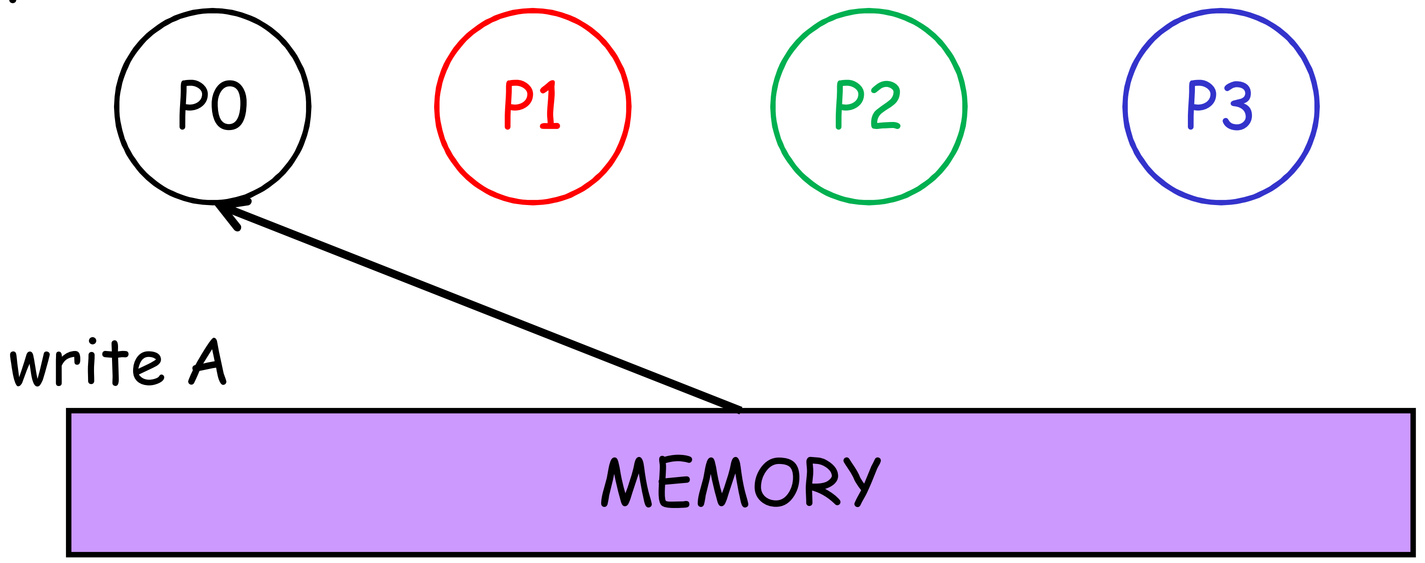
Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
  - Picks the next access from a randomly chosen processor



Sequential consistency

- A legal SC ordering is equivalent to the total order obtained by a randomly moving switch connecting the processors to memory
  - Picks the next access from a randomly chosen processor



Total order: read A, write X, read Y, write A

Sequential consistency

- Lamport's definition of SC
  - A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

What is program order?

- Any legal re-ordering is allowed
- The program order is the order of instructions from a sequential piece of code where programmer’s intuition is preserved
  - The order must produce the result a programmer expects
- Can out-of-order execution violate program order?
  - No. All microprocessors commit instructions in-order and that is where the state becomes visible
  - For modern microprocessors the program order is really the commit order

Example

P0: x=8; u=y; v=9;

P1: r=5; y=4; t=v;

Total order: x=8; u=y; r=5; y=4; t=v; v=9;

Another legal total order:

x=8; r=5; y=4; u=y; v=9; t=v;

- All such total orders are allowed by sequential consistency
  - A multiprocessor implementing sequential consistency can produce an output conforming to any of these total orders
  - All these possible outputs are correct for a sequentially consistent multiprocessor

OOO and SC

- Can out-of-order (OOO) execution violate SC?
  - Yes. Need extra logic to support SC on top of OOO
- Consider a simple example (all variables are zero initially)

  P0: x=w+1; r=y+1;

  P1: y=2; w=y+1;

  - Instructions commit in order with x=4, r=1, y=2, w=3
  - How is that possible?

OOO and SC

P0: x=w+1; r=y+1;

P1: y=2; w=y+1;

– Instructions commit in order with x=4, r=1, y=2, w=3

- Suppose the load that reads w takes a miss and so w is not ready for a long time; therefore, x=w+1 cannot complete immediately; w returns with value 3
- Inside the microprocessor r=y+1 completes (but does not commit) before x=w+1 and gets the old value of y (possibly from cache); eventually instructions commit in order with x=4, r=1, y=2, w=3

OOO and SC

P0: x=w+1; r=y+1;

P1: y=2; w=y+1;

– Instructions commit in order with x=4, r=1, y=2, w=3

- Proving that this is not legal under SC
  – We have the following partial orders

  P0: x=w+1 < r=y+1 and P1: y=2 < w=y+1

  Cross-thread: w=y+1 < x=w+1 and r=y+1 < y=2

  – Combine these to get a contradictory total order

  x=w+1 < r=y+1 < y=2 < w=y+1 < x=w+1

  Need special support to guarantee SC in the presence of OOO instruction execution

  Note that the system is still coherent

SC example

- Consider the following example

  P0: A=1; print B;

  P1: B=1; print A;

- Possible outcomes for an SC machine
  - (pA, pB) = (0,1); interleaving: B=1; print A; A=1; print B
  - (pA, pB) = (1,0); interleaving: A=1; print B; B=1; print A
  - (pA, pB) = (1,1); interleaving: A=1; B=1; print A; print B OR A=1; B=1; print B; print A
  - (pA, pB) = (0,0) is impossible

Implementing SC

- Two basic requirements
  - Memory operations issued by a processor must become visible to others in program order
  - Need to make sure that all processors see the same total order of memory operations: in the previous example for the (0,1) case both P0 and P1 should see the same interleaving: B=1; print A; A=1; print B

Implementing SC

- The tricky part is to make sure that writes become visible in the same order to all processors
  - Write atomicity: as if each write is an atomic operation
  - Otherwise, two processors may end up using different values (which may still be correct from the viewpoint of cache coherence, but will violate SC)

Write atomicity

- Example (A=0, B=0 initially)

  P0: A=1; P1: while (!A); B=1;

  P2: while (!B); print A;

- A correct execution on an SC machine should print A=1
  - A=0 will be printed only if write to A is not visible to P2, but clearly it is visible to P1 since it came out of the loop
  - Thus A=0 is possible if P1 sees the order A=1 < B=1 and P2 sees the order B=1 < A=1 i.e. from the viewpoint of the whole system the write A=1 was not "atomic"
  - Without write atomicity P2 may proceed to print 0 with a stale value from its cache

Summary of SC

- Program order from each processor creates a partial order among memory operations
- Interleaving of these partial orders defines a total order
- Sequential consistency: one of many total orders
- A multiprocessor is said to be SC if all executions on this machine are SC compliant

Implementation of SC

- MIPS R10000 implements SC
  - OOO execution is allowed, and a proper recovery mechanism is invoked when a potential violation of total order is detected
    - The approach is somewhat conservative because the detection mechanism is far from ideal
  - A violation in the total order manifests through "wrong" values supplied by loads
    - Loads execute potentially long before they commit
    - The value supplied by a load when it executes may get changed by some other thread before the load commits
    - If the load is allowed to commit the "old" value, this may violate SC (note that this is still coherent)
      - We have already seen an example

Implementation of SC

- MIPS R10000 implements SC
  - How to detect such loads?
  - How to recover from such a violation?
    - Consider a load to address A in thread X
    - The load executes at time t1 and gets a value v
    - The load is yet to commit and stays in the ROB of the processor running X
    - Another thread Y performs a store to address A with a new value v' at time t2
    - At this point, an invalidation for A is sent to X
    - The invalidation searches the load queue for all matching loads; if found, all instructions are cancelled starting from the oldest such load; these instructions will be refetched and re-executed

Implementation of SC

- MIPS R10000 implements SC
  - Need to be careful about store commit
    - Tempting to remove an in-flight store from the head of ROB to allow following instructions to commit
    - This may violate SC

    T0: A=1; print B;

    T1: B=1; print A;
  - MIPS R10000 keeps stores in ROB until they are complete

Relaxed models

- Implementing SC requires complex hardware and verification effort
  - But such violations are rare
  - 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

    T0: A=1; B=1; flag=1; T1: while (!flag); print A; print B;

    - Can re-order the stores to A and B without any problem
    - Cannot compromise on precise exception, however
  - SC is too restrictive; relaxing it does not always violate programmers' intuition

Relaxed models

- Three attributes of a relaxed model
  - 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 (often SC-compliant order is required)
  - Programmer's interface: set of rules, if followed, will lead to an execution as expected by the programmer; specified in terms of high-level language annotations and labels
  - Translation mechanism: how to translate programmer's annotations to hardware actions

Total store order (TSO)

- One of the many relaxed models
- TSO allows a load to bypass and commit earlier than an older incomplete store
  - A blocked store at the head of the ROB can be removed (but remains in a FIFO store buffer) and subsequent instructions are allowed to commit bypassing the blocked store
  - Motivation: can hide latency of store operations
  - This is the only allowed re-ordering
    - Only a load can bypass an older store
    - A load cannot bypass an older load; a store cannot bypass an older load; a store cannot bypass an older store

Total store order (TSO)

- TSO allows a load to bypass and commit earlier than an older incomplete store
  - As usual, precise exception must be guaranteed meaning that a store at the head of the ROB can unblock the ROB only when it is guaranteed that the store cannot raise an exception
    - Easy to ascertain because a store can raise only page fault exceptions
    - A store is cleared of page fault exception as soon as it passes TLB lookup

Total store order (TSO)

- TSO allows a load to bypass and commit earlier than an older incomplete store
  - 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; P1: while (!A); B=1; P2: while (!B); print A;   [same as SC]
  - P0: A=1; print B; P1: B=1; print A; [violates SC]
  - Implemented in many Sun (Oracle) UltraSPARC microprocessors

Total store order (TSO)

- How to force an SC-compliant outcome on a multiprocessor implementing TSO?
  - Required when porting a program from, say, MIPS R10000 to a SPARC processor platform
  - Must ensure that a load cannot bypass an older store
  - 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, store-to-load fence just before the load instruction

Total store order (TSO)

- How to force an SC-compliant outcome on a multiprocessor implementing TSO?

  P0: A=1; membar #storeload; print B;

  P1: B=1; membar #storeload; printA;

  - The fence instruction will not allow graduation of a load until all stores before it graduates
  - If a fence instruction is not available, substituting the load by a read-modify-write (e.g., *ldstub* in SPARC) also works; a read-modify-write contains a store and hence, it cannot be re-ordered before an older store (that would violate TSO)