Memory Consistency Models
Memory Consistency Models

- Formal specification of how the memory system will appear to the programmer

- Places restrictions on the value that can be returned by a “read” operation in a shared memory program execution
  - “Read” should return the value of the last “Write” to the same location
  - For uniprocessor, “last” is defined by program order
  - For a multiprocessor, not clear how to define “last write”

- Example code in Figure 1 of tech report

- Why do we care?
  - Programmability
  - Performance
  - Portability
Sequential Consistency

An extension of uniprocessor “program order”

A multiprocessor is sequentially consistent if

- Result of any execution is the same as if the operation of all processors were executed in some sequential order
- Operation of each processor appear in this sequence in the order specified by its program

Advantage

- Simple and intuitive programming model

Disadvantages

- Prevents many hardware optimizations (e.g., write buffers)
- Prevents many compiler optimizations (e.g., code motion)

Example programs: Figure 4
Implementing Sequential Consistency

- Need to maintain two requirements
  - Program order
  - Atomicity for memory operations
- SC restricts some common optimizations, even in the absence of caches
- Architectures without caches
  - Write Buffers with bypassing (Fig 5a)
  - Overlapping write operations (Fig 5b)
  - Non-blocking read operations (Fig 5c)
Implementing Sequential Consistency (Cont.)

- Architectures with caches
  - Cache coherence represents the mechanism that propagates a newly written value to the cached copies of the modified location
  - Memory consistency model is the policy that places an early and late bound on when a new value can be propagated to any given processor
  - How do we detect the completion of a write operation?

- Write atomicity
  - Writes to the same location need to be serialized (Figure 6)
  - Prevent read from returning new value until all acknowledgements for write are received (Figure 4b)
Relaxed Memory Models

- Can relax either:
  - Program order requirement
  - Write atomicity requirement

- Relaxing program order requirement
  - Write to a following read
  - Two writes
  - Read to a following read or write

- Relaxing write atomicity requirement
  - Can a read return the value of another processor’s write before the write is visible to all processors?

- Relaxing both requirements
  - Can a processor read the value of its own previous write before it is made visible to all other processors?

- Figure 8 summarizes all relaxed models
- Figure 9 shows some example systems with relaxed models
Relaxing Write to Read Program Order

- IBM 370
- Total Store Order (TSO), implemented in SPARC V8
- Processor Consistency (PC)
- All techniques allow a read to be reordered wrt previous writes from the same processor
  - Enable write buffers
- Techniques differ on when to allow a read to return the value of a write (Figure 8)
- Figure 10 shows example of how techniques are different
Relaxing Write to Read & Write to Write Program Orders

- Partial Store Order (PSO), implemented in SPARC V8
- Writes to different locations from the same processor can be pipelined or overlapped
  - Writes allowed to reach memory or other caches out of program order
- A processor can read the value of its own write early
- A processor is prohibited from reading another processor’s write until it is visible to all other processors
Relaxing All Program Orders

- Relax program order between all operations to a different location
- A read or write may be reordered wrt a following read or write to a different location
- Allows non-blocking reads (lockup-free caches, speculative execution)
- Allows almost all compiler optimizations

Examples
- Weak Ordering (WO)
- Release Consistency (RCpc, PCsc)
- DEC Alpha
- PowerPC
- Relaxed Memory Order (RMO) in SPARC V9
Weak Ordering (WO)

- Classifies memory operations into two categories
  - Data operations
  - Synchronization operations
- To enforce program order between two operations, programmer needs to specify synchronization operation
- Intuition: reordering data operations in between synchronization operations would not affect correctness
- Writes appear atomic to programmer
Release Consistency (RC)

Classifies memory operations into:
- Ordinary operations
- Special operations
  - **Sync**: Synchronization operations
  - **Nsync**: asynchronous data operations, not used for synchronization

Sync operations are either
- **Acquire**: read operation to gain access to a set of shared locations (e.g., lock, spin for a flag to be set)
- **Release**: write operation to grant permission for accessing set of shared location (e.g., unlock, set flag)

Different RC Models provide different program orders among special operations
- **RCsc**: acquire → all, all → release, special → special
- **RCpc**: RCsc: acquire → all, all → release, special → special except for special write followed by a special read
Reading Assignment

- Midterm on Monday
  - Open notes, books, calculator
  - No sharing of notes (bring your own)
  - No laptops, cell phones, PDAs

- Wednesday