Memory Coherence and Consistency

Reading for this Module

- Memory Coherence
  - Snooping bus protocols
    - Section 6.3 and 6.4
  - Directory protocols
    - Section 6.5 and Section 6.6

- Memory Consistency
  - Section 6.8
Additional processors are used to improve performance

We adopt a simplified model wherein each processor executes a distinct thread

Threads share data

Handling Shared Data

Multiple cache copies exist during read-only sharing

Intuitively we must ensure that the most “recent” value of a shared variable is read → coherency!
Cache Coherency

- Fundamentally this is a synchronization problem
  - Synchronized updates to shared variables

- A memory system is coherent if
  - Program order for loads/stores are preserved
  - All stores eventually become visible
  - Write serialization: all processors see the same order of writes to a variable

Approaches

- Software schemes
  - Do not cache shared data

- Invalidation-based protocols
  - One processor "owns" the shared data, all others invalidate on writes

- Update-based protocols
  - Writes are broadcast to all processors that share data
  - Caches and memory may be inconsistent
Performance Issues

- Use of memory bandwidth
  - Different protocols make different demands on bandwidth of memory and the bus

- Memory traffic
  - Different protocols produce different levels of bus traffic

- Implementation complexity
  - Hardware complexity of the cache state machines
  - Impact on bus protocol

System Model: Snooping Protocols

- Single physical address space with uniform memory access (UMA) times
- Basic cache operation remains unchanged
- State of a cache line indicates sharing status
  - State is associated with the physical, processor cache line and not with the contents of the line
Cache State Transitions: Based on CPU Requests

INVALID

- CPU Read hit
- CPU Write hit
- CPU read miss: block present in cache
- CPU write miss: block not present in cache

SHARED

- CPU Read hit
- CPU Write hit
- CPU read miss: write-back block, place read on bus
- CPU write miss: write-back cache block, place write miss on bus

EXCLUSIVE

- CPU Read hit
- CPU Write hit
- CPU read miss: write-back block, place read on bus
- CPU write miss: write-back cache block, place write miss on bus

Cache State Transitions: Based on Bus Requests

INVALID

- Write miss for this block
- Read miss for this block: write back block, abort memory access

SHARED

- CPU read miss
- CPU read miss: block not present in cache

EXCLUSIVE

- Write miss for this block
- Read miss for this block: write back block, abort memory access
Observations

- The bus serves an arbitrator for serializing accesses to a shared line
  - Cannot update the cache line unless the bus is obtained

- All caches see the same sequence of writes
  - Local state changes can be orchestrated in a globally consistent fashion

Example
Implementation Issues

- In reality the preceding state transitions are not atomic
  - For example, miss, acquire the bus, and receive a response will not in practice be atomic
  - Split transaction buses introduce non-atomic operations

- Multiple coordinating entities on the same bus

- Interference between snooping and CPU accesses
  - Duplicate the cache tags
  - Use multi-level inclusion for L2/L3 caches

Further Optimizations

- In practice, protocols distinguish between write hits and write misses
  - Utilize the notions of invalidations and "ownership"

- Distinguish the exclusive, consistent state of the cache line
  - Let us refer to this as a clean-private state
  - MESI protocol

- Allow blocks to be shared without writing back
  - Distinguish shared, but dirty state.
A commercial protocol: MESI Protocol

Major Transitions

- Industry standard, invalidation-based protocol for SMPs
- Reading: Find a complete specification as used in the Pentium and understand all of the transitions

ECE 4100/6100 (15)

Scaling Multiprocessors

- A bus is a bottleneck to scaling to a large systems
  - Electrical issues
  - Contention issues
- Goal: scalable memory and interconnection bandwidth
  - Message passing networks for scalable bandwidth
  - Physically distributed memories for scalable memory bandwidth
- Problem: snooping schemes are not scalable

ECE 4100/6100 (16)
The Problem

- Move control to the source → the original copy of the line in memory
- Some cache actions now complete only after interrogation of the source → send/receive messages

How do We Handle Cache Coherence?

- A logical approach is to store the state of each memory block in a centrally accessible directory
  - Each directory entry contains
    - Set of processors sharing a block
    - State of the block, for example, modified
  - The directory can be queried on each memory access
  - Directory size is proportional to product of number of processors and number of memory blocks
  - Distribute the directory among the processor nodes
Using Distributed Directories

- Single physical, distributed address space with non-uniform memory access (NUMA) times
- Basic snooping protocol state machine transitions are preserved

Some Additional Concepts

- Messages are received in the order sent
- Directory entry indicates state of cached blocks and the members of the sharing set
Directory Protocol Features

- The \{sharing set\} is the set of processors with a copy of a memory block
  - Implementation
    - Bit vectors and fully mapped entries
    - Linked lists

- Consistency strategy
  - Shared lines always consistent with home copy

- Notification strategy
  - Invalidation rather than update

The Local Processor State Machine

- CPU read miss: send read miss msg
- CPU read hit
- Fetch
- Invalidate: data write back
- CPU write: send data write msg
- CPU write hit
- CPU write miss: data write back
- CPU read miss
- CPU write: data send write msg
The Directory State Machine

- Note that the state of the memory block refers to the state of the copies in remote caches.

Update vs. Invalidation Strategies

- When using a mapped directory
  - Invalidation messages are replaced by update values

- When using a linked directory
  - Update messages are propagated
  - Requester is added to the head of the list

- Invalidationss reduce the size of the sharing set, while updates increase their size
  - Updates reduce new requests for the line
  - Invalidations increase network traffic
Performance scaling is achieved via the use of multiple processors each working on one part of the application.

Caching of shared data leads to the cache coherency problem – Essentially a synchronization problem.

Solutions depend on the scale of the system:
- Small scale machines using a shared bus → snooping protocols
- Large scale machines using a message passing network → directory based protocols
Memory Consistency Models

Memory Consistency

- What can the programmer assume about the servicing of memory operations?
  - For example, will they occur in program order?
  - Why are these assumptions important?
The Uniprocessor Case

- What is the model of memory behavior?
  - Memory operations occur in program order → read returns the value of the last write in program order

- Semantics defined by sequential program order
  - Simple to reason about but over constrained
    - Really only need to honor control and data dependencies
  - Reality is that independent operations can execute in parallel
  - Optimizations preserve these semantics

The Multiprocessor Case

- A memory consistency model
  - A set of rules governing how the memory systems will process memory operations from multiple processors
  - Contract between the programmer and system
  - Determines what optimizations can be performed for correct programs
Sequential Consistency

[Lamport] “A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program”

Implications of Sequential Consistency

- Program order requirement
  - Note that memory systems may be parallel and that the network between processors and memories may re-order instructions

- Atomicity requirement
  - Informally, a write takes place instantaneously with respect to the ability of all other processors to read it

Program Ordering Issues

- Violation of program order requirement → can lead to incorrect parallel programs
  - Use of write buffers
    - Note: Does not violate data dependence in uniprocessor systems

- Ordering issues arise naturally in systems with caches

- Compiler re-ordering of instructions lead violations of sequential consistency

Atomicity Issues in the Presence of Caches

- Consider the above example and an update based protocol
- Atomicity can be ensured by the following two conditions
  - Writes completion: result of a write cannot be used until all copies have been updated/invalidated
  - A write must be atomic “system wide”
  - Writes to a location are serialized, i.e., all processors see writes to the same location in the same order
Caches Coherency vs. Consistency

- Coherence can be viewed as mechanism to propagate new values (equivalently invalidations) to cached copies
- Coherence requirements typically are
  - All writes to a variable are eventually visible to all processors
  - Serialization of writes to a variable
- Preceding insufficient for sequential consistency which requires serialization of all writes to all locations

Relaxed Memory Models

- Relaxation of memory requirements is achieved along two dimensions
  - Relaxing the program ordering requirement
  - Relaxing the atomicity requirement
- Key idea
  - Let reads and writes complete out of order
  - Use synchronization primitives to enforce ordering constraints
  - The program can behave “as if” it were a sequentially consistent execution
- Goal is increased performance from concurrency in memory operations
Relaxed Memory Models (cont.)

- **Processor Consistency**
  - Writes by any processor are seen by all processors in the order they were issued
  - For any variable, all processors see writes in the same order
  - Weaker than sequential consistency since the same ordering is not guaranteed to be seen by all processors

- **Weak Consistency**
  - Distinguish between data operations and synchronization operations
  - Synchronization operations are sequentially consistent
    - All processors see synchronization operations in the same order
  - When a synchronization operation is issued, the memory pipeline is flushed
    - All pending writes must complete before the synchronization operation executes

Relaxed Memory Models (cont.)

- **Release Consistency**
  - Increases overlap in memory operations restricted by the weak consistency model
    - Similarity with instruction issue and dependences?
The Programmers View

- The use of synchronized programs
  - All accesses to shared data are synchronized
    - Data references are ordered by synchronization primitives
  - Thus these programs are data-race free
    - Outcome does not depend on the relative speed of processors, network, and system software

- Utilize a programmer-centric view of consistency
  - Do not have to reason about ordering and atomicity constraints
  - Write programs to conform to program semantics and let the compiler and system libraries bring optimizations to bear based on the model supported in the language

Summary

- Consistency models are a set of rules that can be relied on by the programmer and compiler
- Consistency Models determine the system optimizations that are possible
  - Overlapping of memory operations from multiple processors
- Many optimizations can violate consistency model semantics
  - Leads to incorrect execution
- Consistency models are distinct from coherence
  - The latter is concerned with updates/invalidations to a single shared variable
  - The former is concerned with the behavior of memory references from multiple concurrent threads
• Be able to distinguish between the various consistency models
  – From the perspective of definition

• Given a set of LD/SD references from multiple threads, be able to state the ordering and atomicity constraints on these LD/SD operations for a consistency model
  – Can the follow sequence of operations happen if the memory system is sequentially consistent?
  – Can the following sequence of operations happen if the memory system adheres to processor consistency?
  – Will this optimization violate sequential consistency?

• Given a sequence of references taking place in a specific order does it satisfy the constraints of a specific model?

• Given the state of lines in multiple caches, what are the states of the cache lines after a given sequence of references
  – For a snooping cache coherence protocol
  – For a directory based protocol

• What is the performance impact of choices such as
  – Invalidation vs. update protocols
  – Line size
  – Write back vs. write through

• Extend each protocol (i.e., extend the state machine) to use a new state: exclusive but unmodified