Coherence vs. Consistency

- Intuition says load should return latest value
  - What is latest?

- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations

- A memory system is coherent if, for all locations,
  - Can serialize all operations to that location such that,
    - Operations performed by any processor appear in program order
      - Program order = order defined by program text or assembly code
    - A read gets the value written by last store to that location

Why Consistency is Important

- Consistency model defines correct behavior
  - It is a contract between the system and the programmer
  - Analogous to the ISA specification

- Coherence protocol is only a means to an end
  - Coherence is not visible to software
  - Enables new system to present same consistency model despite using newer, fancier coherence protocol
  - Systems maintain backward compatibility for consistency (like ISA)

- Consistency model restricts ordering of loads/stores
  - Does NOT care at all about ordering of coherence messages
Why Coherence ≠ Consistency

/* initially, A = B = flag = 0 */

\[\begin{align*}
P_1 & : A = 1; \quad \text{while (flag == 0); /* spin */}

P_2 & : B = 1; \quad \text{print A;}

\text{flag = 1; \quad print B;}
\end{align*}\]

- Intuition says we should print \(A = B = 1\)
- Yet, in some consistency models, this isn’t required!
- Coherence doesn’t say anything ... why?

Sequential Consistency

- Leslie Lamport 1979:
  “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”

Abstraction: a multitasking uniprocessor

The Memory Model

SC: Definitions and Sufficient Conditions

- Sequentially consistent execution
  - Result is same as one of the possible interleavings on uniprocessor

- Sequentially consistent system
  - Any possible execution corresponds to some possible total order

- Alternate equivalent definition of SC
  - There exists a total order of all loads and stores (across all processors), such that the value returned by each load equals the value of the most recent store to that location
Definitions

• Memory operation
  – Load, store, atomic read-modify-write to mem location

• Issue
  – An operation is issued when it leaves processor and is presented to
    memory system (cache, write-buffer, local and remote memories)

• Perform
  – A store is performed wrt to a processor P when a load by P returns value
    produced by that store or a later store
  – A load is performed wrt to a processor when subsequent stores cannot
    affect value returned by that load

• Complete
  – A memory operation is complete when performed wrt all processors.

• Program execution
  – Memory operations for specific run only (ignore non-memory-referencing
    instructions)

Sufficient Conditions for Sequential Consistency

• Processors issue memory ops in program order

• Processor must wait for store to complete before
  issuing next memory operation

• After load, issuing proc waits for load to complete,
  and store that produced value to complete before
  issuing next op

• Easily implemented with shared (physical) bus
  This is sufficient, but more than necessary

SGI Origin: Preserving Sequential Consistency

• MIPS R10000 is dynamically scheduled
  – Allows memory operations to issue and execute out of program order
  – But ensures that they become visible and complete in order
  – Doesn’t satisfy sufficient conditions, but provides SC

• An interesting issue w.r.t. preserving SC
  – On a write to a shared block, requestor gets two types of replies:
    • Exclusive reply from the home, indicates write is serialized at
      memory
    • Invalidation acks, indicate that write has completed wrt processors
  – But microprocessor expects only one reply (as in a uniprocessor)
  – So replies have to be dealt with by requestor’s HUB
  – To ensure SC, Hub must wait until inval acks are received before
    replying to proc
    • Can’t reply as soon as exclusive reply is received
    • Would allow later accesses from proc to complete
      (writes become visible) before this write

Outline

• Difference Between Coherence and Consistency

• Sequential Consistency

• Relaxed Memory Consistency Models
  – Motivation
  – Processor Consistency
  – Weak Ordering & Release Consistency

• Consistency Optimizations

• Synchronization Optimizations
**Why Relaxed Memory Models?**

- **Motivation with directory protocols**
  - Misses have longer latency
  - Collecting acknowledgments can take even longer

- **Recall SC requires strict ordering of reads/writes**
  - Each processor generates a local total order of its reads and writes
  - All local total orders are interleaved into a global total order

- **Relaxed models relax some of these constraints**
  - PC: Relax ordering from writes to (other processors') reads
  - RC: Relax all read/write orderings (but add "fences")

**Processor Consistency (PC):**

```
/* initially, A = B = 0 */
P1, P2
A = 1; B = 1
r1 = B; r2 = A;
```

- **Processor Consistency (PC)**
  - Allows r1==r2==0 (not allowed by SC)
  - Examples: IBM 370, Sun Total Store Order (TSO), & Intel IA-32

- **Why do this?**
  - Allows FIFO write buffers → performance!
  - Does not confuse programmers (too much)

---

**Write Buffers w/ Read Bypass**

```
P1
Write Flag 1
Read  Flag 2
i1

P2
Read Flag 1
Write Flag 2
i2

Shared Bus

Flag 1: 0
Flag 2: 0
```

- **P1**
  - Flag 1 = 1
  - if (Flag 2 == 0)
  - critical section

- **P2**
  - Flag 2 = 1
  - if (Flag 1 == 0)
  - critical section

---

**Also Want “Causality” (Transitivity)**

```
/* initially all 0 */
P1, P2, P3
A = 1; while (flag1==0) {} while (flag2==0) {};
flag1 = 1; flag2 = 1; r3 = A;
```

- **We expect P3’s r3=A to get value 1**
- **All commercial versions of PC guarantee causality**
So Why Not Relax All Order?

/* initially all 0 */

P1
A = 1;
B = 1;
flag = 1;

P2
while (flag == 0); /* spin */
r1 = A;
r2 = B;

- We'd like to be able to reorder "A = 1"/"B = 1" and/or "r1 = A"/"r2 = B"
  - Useful because it could allow for OOO processors, non-FIFO write buffers, delayed directory acknowledgments, etc.
- But, for sanity, we still would like to order
  - "A = 1" / "B = 1" before "flag =1"
  - "flag = 0" before "r1 = A" / "r2 = B"

Order with “Synch” Operations

/* initially all 0 */

P1
A = 1;
flag = 1;

P2
while (SYNCH flag == 0);
B = 1;
r1 = A;
r2 = B;

- Called weak ordering (WO) or “weak consistency”
- SYNCH orders all prior and subsequent operations
- Alternatively, release consistency (RC) specializes
  - Acquire: forces subsequent reads/writes after
  - Release: forces previous reads/writes before

Weak Ordering Example

Release Consistency Example
Review: Directory Example with Sequential Consistency

Directory Example with Release Consistency

Commercial Models Use “Fences”

```
/* initially all 0 */
P1 A = 1; while (flag == 0);
B = 1; FENCE;
FENCE; r1 = A;
flag = 1; r2 = B;
```

Examples: Compaq Alpha, IBM PowerPC, & Sun RMO
- Can specialize fences (e.g., RMO)

Intel IA-64 is RCpc (acquires & releases obey PC)

The Programming Interface

- WO and RC require synchronized programs

- All synchronization operations must be labeled and visible to the hardware
  - Easy (easier!) if synchronization library used
  - Must provide language support for arbitrary Ld/St synchronization
    (event notification, e.g., flag)

- Program written for weaker model OK on stricter
  - E.g., SC is a valid implementation of TSO, WO, or RC
Outline

- Difference Between Coherence and Consistency
- Sequential Consistency
- Relaxed Memory Consistency Models
- Consistency Optimizations
- Synchronization Optimizations

Consistency is an Abstraction

- We only have to implement a system that preserves the illusion of the specified consistency model
  - Like OOO processor preserves illusion of in-order execution
- We can speculate and recover if speculation leads to violation of consistency model
  - E.g., MIPS R10000 processor speculates on consistency
- Scheurich’s Optimization (for directory protocols)
  - Immediately acknowledge incoming invalidation (to Shared block)
  - Yet continue to read this block (i.e., pretend Inv didn’t arrive yet)
  - Must invalidate before asking mem system for upgrade to any block
  - Optimization does NOT even violate SC

Scheurich’s Optimization – Legal Example

```
/* initially A=B=0, P1 owns A, P2 shares B */
```

```
P1
  issue GETX B
  ST B=1
  ST A=1

P2
  LD B=0
  Ack Inv B
  LD B=0 /* Legal! */
```

Scheurich’s Optimization – Illegal Example

```
/* initially A=B=0, P1 owns A, P2 shares B */
```

```
P1
  issue GETX B
  ST B=1
  ST A=1

send A=1
```

```
P2
  LD B=0
  Ack Inv B
  LD B=0 /* Legal! */
```

```
issue GETS A
```

```
LD A=1
LD B=0 /* Illegal */
```
Outline

• Difference Between Coherence and Consistency
• Sequential Consistency
• Relaxed Memory Consistency Models
• Consistency Optimizations (Wed more on this)

• Synchronization Optimizations
  – Review of Synchronization (Section 5.5 of Culler/Singh)
  – Synchronization Design Tradeoffs (More Section 5.5)
  – Cool New Synchronization Schemes (e.g., SLE)

Synchronization

• Mutual exclusion (critical sections)
  – Lock & unlock

• Event notification
  – Point-to-point (producer-consumer, flags)
  – Global (barrier)

• Locks, Barriers, Flags, etc.
  – How are these implemented?

Anatomy of A Synchronization Operation

1) Acquire method
   • Method for trying to obtain lock or proceed past barrier

2) Waiting algorithm
   • Spin or busy wait
   • Block (suspend)

3) Release method
   • Method to allow other processes to get past synchronization event

HW/SW Implementation Tradeoffs

• User wants high level (ease of programming)
  – LOCK(lock_variable), UNLOCK(lock_variable)
  – BARRIER(barrier_variable, Num_Procs)

• Hardware’s advantage
  – The Need for Speed (it’s fast)

• Software’s advantage
  – Flexibility

• Goals
  – Low latency
  – Low traffic
  – Scalability
  – Low storage overhead
  – Fairness
How NOT To Implement Locks

- LOCK
  
  while (lock_variable == 1); // spin
  
  lock_variable = 1;

- UNLOCK
  
  lock_variable = 0;

- Implementation requires Mutual Exclusion!
  – Can have two processes successfully acquire the lock

- Mutual exclusion requires atomic operations ...

Atomic Read-Modify-Write Operations

- Test&Set(r,x)
  
  r = m[x]
  m[x] = 1

- Swap(r,x)
  
  r = m[x], m[x] = r

- Compare&Swap(r1,r2,x)
  
  if (r1 == m[x]) then
    r2 = m[x], m[x] = r2

- Fetch&Op(r,x,op)
  
  r = m[x], m[x] = op(m[x])

Aside: Load-Locked Store-Conditional

- Pair of instructions that can implement lots of atomic operations
- Load-Locked loads address A into register r1
- We can then manipulate r1 for a while
- Store-Conditional writes r1 back to A if A hasn’t been modified
  – Invalidation
  – Replacement
  – Context switch

lock: l1 r1, memloc // load-lock memloc into r1
  sc memloc, #1 // conditionally store 1 into memloc
  beqz lock // if sc fails, try lock again
  ret

unlock: st location, #0
  ret

Performance of Test & Set

- Problem 1: Bad performance under contention
  – Test&Set causes GetExclusive coherence request
- Problem 2: Not fair
  – Processors don’t get lock in order in which they request it

Both problems due to the waiting algorithm!
Better Lock Implementations

- Two choices:
  - Don’t execute test&set so much \( \rightarrow \) test&set with back-off
  - Spin without generating bus traffic \( \rightarrow \) test-and-test&set

- Test&Set with back-off
  - Insert delay between test&set operations (not too long)
  - Exponential seems good \( (k^c) \)
  - Not fair

- Test-and-Test&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue test&set
  - Intuition: No point in trying to set the location until we know that it’s not set, which we can detect when it gets invalidated
  - Still contention after invalidation
  - Still not fair

Lock Implementation Details

- To Cache or Not to Cache, that is the question

  Uncached
  - Latency for one operation increases
  - Fast hand-off between processes

  Cached
  - Might generate a lot of traffic if lock moves around
  - Might reuse lock a lot (locality), then traffic would be reduced by caching

- Must keep ownership for entire read-modify-write cycle
  - Synchronization operation is visible to the memory system

Fetch&Increment Based Locks

- Ticket Lock (like at the bakery)
  - LOCK
    - Obtain number via fetch&inc
    - Spin on now-serving counter
  - UNLOCK
    - Increment now-serving counter

- Array based Lock
  - Obtain location to spin on rather than value
  - Fair
  - Slight increase in storage
  - Put locations in separate cache blocks, else same traffic as t&t&s

H/W Queue-Based Locks: QOLB

- Distributed directory (SCI)
- Lock bit in every cache line
- Maintains sharing list in a doubly-linked list
- Local spinning in the cache: cache hit while waiting
S/W Queue-Based Locks: MCS

• To emulate QOLB in software:
  • (master) lock resides in shared memory
  • Requesters allocate a block in shared memory to spin on
    – The flag the requester spins on is originally set to 1
  • Upon acquire, requester inserts its block in linked list
    – It reads the address of the last requester’s local block
    – It writes its local block address to the master
  • Upon release
    – Releaser removes itself from the linked list
    – Sets the next processor in line’s flag to 0

Mechanisms To Reduce Lock Overhead

• Optimizations used by locks:
  – Local spinning (on data in local cache)
  – Queue-based locking
  – Collocation of lock and the data it protects
  – Synchronous prefetch

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>L-spin</th>
<th>Queue Collocation</th>
<th>S-fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>no</td>
<td>no</td>
<td>optional no</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>yes</td>
<td>no</td>
<td>optional no</td>
</tr>
<tr>
<td>MCS</td>
<td>yes</td>
<td>yes</td>
<td>partial no</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>yes</td>
<td>optional yes</td>
</tr>
</tbody>
</table>

Microbenchmark Analysis

Lock performance with increase in processors

What are the Performance Issues?

• Low latency:
  – Should be able to get a free lock quickly

• Scalability:
  – Should perform well beyond a small number of processors (< 64)

• Low storage overhead

• Fairness

• Blocking/Non-blocking
Performance of Locks

• Performance depends on many factors
  – Amount of contention
  – Number of processors
  – Snooping vs. directory
  – Hardware support for locks

• Test&set is good with no contention
• Array based (Queue) is best with high contention

• Reactive Synchronization by Lim & Agarwal
  – Choose lock implementation based on contention

Point-to-Point Event Synchronization

• Often use normal variables as flags
  \[ a = f(x); \quad \text{while (flag == 0);} \]
  \[ \text{flag} = 1; \quad b = g(a); \]

• If we know \( a=0 \) beforehand
  \[ a = f(x) \quad \text{while (a == 0);} \]
  \[ b = g(a); \]

• Assumes Sequential Consistency!!
  • Full/Empty bits are similar to flags
    – Set on Write
    – Cleared on Read
    – Can’t write if set, can’t read if clear

Implementing a Centralized Barrier

```c
BARRIER(bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0) {
        bar_name.flag = 0;
        bar_name.counter++;
    }
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = 1;
    } else
        while(bar_name.flag == 0) {} /* busy wait */
}
```

But what if P1 doesn’t see flag set to 1 in first barrier before P2 re-enters barrier and resets flag to 0?

Barrier With Sense Reversal

```c
BARRIER(bar_name, p) {
    local_sense = !local_sense; /* toggle private state */
    LOCK(bar_name.lock);
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = local_sense;
    } else
        while(bar_name.flag != local_sense) {} /* busy wait*/
}
```
Outline

• Difference Between Coherence and Consistency
• Sequential Consistency
• Relaxed Memory Consistency Models
• Consistency Optimizations
• Synchronization Optimizations