18-447: Computer Architecture
Lecture 25: Multiprocessor Correctness and Cache Coherence

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2012, 4/25/2012
Reminder: Lab Assignment 7

- Cache coherence in multi-core systems
  - MESI cache coherence protocol

- Due May 4

- Extra credit: Improve the protocol (open-ended)
Final Exam

- May 10

- Comprehensive (over all topics in course)

- Three cheat sheets allowed

- We will have a review session (stay tuned)

- Remember this is 30% of your grade
  - I will take into account your improvement over the course
  - Know the previous midterm concepts by heart
Readings for Today

- **Required cache coherence readings:**
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- **Recommended:**
Last Lecture

- Wrap up runahead execution
  - Benefits and limitations

- Multiprocessing fundamentals
Today

- Bottlenecks in parallel processing
- The memory ordering problem and one solution
  - Sequential consistency
- The cache coherence problem and some solutions
  - MESI protocol (aka the Illinois protocol)
Multiprocessors and Issues in Multiprocessing
MIMD Processing Overview
Review: MIMD Processing

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
  - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization
Review: Main Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations

- Cache consistency
  - More commonly called cache coherence

- Ordering of memory operations
  - What should the programmer expect the hardware to provide?

- Resource sharing, contention, partitioning
- Communication: Interconnection networks
- Load imbalance
Amdahl’s Law

- $f$: Parallelizable fraction of a program
- $N$: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$


- **Maximum speedup limited by serial portion:** Serial bottleneck
- **Parallel portion is usually not perfectly parallel**
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among $N$ processors)
Sequential Bottleneck

![Graph showing speedup vs. parallel fraction for different N values: N=10, N=100, N=1000.](image)

- **Speedup** on the y-axis.
- **f (parallel fraction)** on the x-axis.
- Three curves for different N values, each labeled as N=10, N=100, N=1000.
Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)
  
  ```
  for (i = 0; i < N; i++)
  ```

- Single thread prepares data and spawns parallel tasks (usually sequential)
Another Example of Sequential Bottleneck

InitPriorityQueue(PQ);
SpawnThreads();

ForEach Thread:

while (problem not solved)

Lock (X)
SubProblem = PQ.remove();
Unlock(X);

Solve(SubProblem);
If(problem solved) break;
NewSubProblems = Partition(SubProblem);

Lock(X)
PQ.insert(NewSubProblems);
Unlock(X)

PrintSolution();
Bottlenecks in Parallel Portion

- **Synchronization**: Operations manipulating shared data cannot be parallelized
  - LOCKS, mutual exclusion, barrier synchronization
  - **Communication**: Tasks may need values from each other
    - Causes thread serialization when shared data is contended

- **Load Imbalance**: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
  - Reduces speedup in parallel portion

- **Resource Contention**: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
  - Additional latency not present when each task runs alone
Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - “Embarrassingly parallel” applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of parallel computer architecture is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer’s job easier in writing correct and high-performance parallel programs
Memory Ordering in Multiprocessors
Ordering of Operations

- Operations: A, B, C, D
  - In what order should the hardware execute (and report the results of) these operations?

- A contract between programmer and microarchitect
  - Specified by the ISA

- Preserving an “expected” (more accurately, “agreed upon”) order simplifies programmer’s life
  - Ease of debugging; ease of state recovery, exception handling

- Preserving an “expected” order usually makes the hardware designer’s life difficult
  - Especially if the goal is to design a high performance processor: Load-store queues in out of order execution
Memory Ordering in a Single Processor

- Specified by the von Neumann model
- Sequential order
  - Hardware *executes* the load and store operations *in the order specified by the sequential program*

- Out-of-order execution does not change the semantics
  - Hardware *retires* (reports to software the results of) the load and store operations *in the order specified by the sequential program*

- Advantages: 1) Architectural state is precise within an execution. 2) Architectural state is consistent across different runs of the program \( \Rightarrow \) Easier to debug programs
- Disadvantage: Preserving order adds overhead, reduces performance
Memory Ordering in a Dataflow Processor

- A memory operation executes when its operands are ready

- Ordering specified only by data dependencies

- Two operations can be executed and retired in any order if they have no dependency

- Advantage: Lots of parallelism $\rightarrow$ high performance

- Disadvantage: Order can change across runs of the same program $\rightarrow$ Very hard to debug
Memory Ordering in a MIMD Processor

- Each processor’s memory operations are in sequential order with respect to the “thread” running on that processor (assume each processor obeys the von Neumann model)

- Multiple processors execute memory operations concurrently

- How does the memory see the order of operations from all processors?
  - In other words, what is the ordering of operations across different processors?
Why Does This Even Matter?

- **Ease of debugging**
  - It is nice to have the same execution done at different times have the same order of execution

- **Correctness**
  - Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?

- **Performance and overhead**
  - Enforcing a strict “sequential ordering” can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)
\[ P_1 \quad P_2 \]

\[
\begin{align*}
F_1 &= \emptyset \\
\{ \text{Critical section} \} \\
F_2 &= \emptyset \\
\{ \text{Critical section} \}
\end{align*}
\]

A: \( F_1 = 1 \)  
B: \( \text{IF } (F_2 = \emptyset) \text{ THEN } \{ \text{Critical section} \} \)
C: \( F_1 \neq \emptyset \)  
ELSE \( \{ \ldots \} \)

X: \( F_2 = 1 \)  
Y: \( \text{IF } (F_1 = \emptyset) \text{ THEN } \{ \text{Critical section} \} \)
Z: \( F_2 = \emptyset \)  
ELSE \( \{ \ldots \} \)

Only \( P_1 \) or \( P_2 \) should be in the critical section at any given time, not both.
A Question

- Can the two processors be in the critical section at the same time given that they both obey the von Neumann model?

- Answer: yes
An Incorrect Result (due to an implementation that does not provide sequential consistency)

At time 0:
- $P_1$ executes $A$ (set $F_1 = 1$)
- $P_2$ executes $X$ (set $F_2 = 1$)
- $A$ is sent to memory
- $X$ is sent to memory

$P_1$'s view:
- $F_1$ is complete

$P_2$'s view:
- $F_2$ is complete
Both Processors in Critical Section

\(t_{\text{me 0}}: \begin{align*}
P_1 & \text{ executes } A \\
& \text{(set } F_1 = 1) \text{ st } F_1 \text{ complete } \text{(from } P_1 \text{'s view)} \\
& A \text{ is sent to memory} \\
\end{align*} \quad \begin{align*}
P_2 & \text{ executes } X \\
& \text{(set } F_2 = 1) \text{ st } F_2 \text{ complete } \text{(from } P_2 \text{'s view)} \\
& X \text{ is sent to memory} \\
\end{align*} \)

\(t_{\text{me 1}}: \begin{align*}
P_1 & \text{ executes } B \\
& \text{(test } F_2 = 0) \text{ ld } F_2 \text{ stored} \\
& B \text{ is sent to memory} \\
\end{align*} \quad \begin{align*}
P_2 & \text{ executes } Y \\
& \text{(test } F_1 = 0) \text{ ld } F_1 \text{ stored} \\
& Y \text{ is sent to memory} \\
\end{align*} \)

\(t_{\text{me 50}}: \begin{align*}
& \text{Memory sends back to } P_1 \\
& F_2 (0) \text{ ld } F_2 \text{ complete} \\
\end{align*} \quad \begin{align*}
& \text{Memory sends back to } P_2 \\
& (F_1 (0) \text{ ld } F_1 \text{ complete} \\
\end{align*} \)

\(t_{\text{me 51}}: \begin{align*}
& P_1 \text{ is in critical section} \\
\end{align*} \quad \begin{align*}
& P_2 \text{ is in critical section} \\
\end{align*} \)

\(t_{\text{me 100}}: \begin{align*}
& \text{Memory completes } A \\
& F_1 = 1 \text{ in memory } \text{(too late!)} \\
\end{align*} \quad \begin{align*}
& \text{Memory completes } X \\
& F_2 = 1 \text{ in memory } \text{(too late!)} \\
\end{align*} \)
What happened?

<table>
<thead>
<tr>
<th>P₁'s view of mem. ops</th>
<th>P₂'s view</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ((F₁=1))</td>
<td>X ((F₂=1))</td>
</tr>
<tr>
<td>B ((\text{test } F₂=0))</td>
<td>Y ((\text{test } F₁=0))</td>
</tr>
<tr>
<td>X ((F₂=1))</td>
<td>A ((F₁=1))</td>
</tr>
</tbody>
</table>

B executed before X

Y executed before A

Problem!

These two processors did not see the same order of operations in memory.
How Can We Solve the Problem?

Sequential consistency:

All processors see the same order of operations to memory.

i.e., all operations happen in an order (global total order) that is consistent across all processors.

Assumption: Within this global order, each processor's operations appear in sequential order with respect to its own operations.
Sequential Consistency


- A multiprocessor system 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

- This is a memory ordering model, or memory model
  - Specified by the ISA
Programmer’s Abstraction

- Memory is a switch that services one load or store at a time for any processor
- All processors see the currently serviced load or store at the same time
- Each processor’s operations are serviced in program order
Sequentially-Consistent Operation Orders

\[ \begin{align*}
A &\quad A &\quad A &\quad X &\quad X &\quad X \\
B &\quad X &\quad X &\quad A &\quad A &\quad X \\
X &\quad B &\quad Y &\quad B &\quad A \\
Y &\quad Y &\quad B &\quad Y &\quad B &\quad B
\end{align*} \]

potential correct global orders
(all are correct)
Consequences of Sequential Consistency

1. Within the same execution, all processors see the same global order of operations to memory → No correctness issue

2. Across executions, different global orders can be observed (each of them is sequentially consistent) → Debugging is still difficult as order changes across executions
Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques

- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model
  - How about enforcing a global order only at the boundaries of synchronization?
    - Relaxed memory models
    - Acquire-release consistency model
Issues with Sequential Consistency?

- Performance enhancement techniques that could make SC implementation difficult

- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores

- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors
Cache Coherence
Shared Memory Model

- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two
- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?
Cache Coherence

- **Basic question:** If multiple processors cache the same block, how do they ensure they all see a consistent state?
The Cache Coherence Problem

P1

Interconnection Network

Main Memory

P2

ld r2, x

1000

1000
The Cache Coherence Problem

ld r2, x

P1

1000

P2

ld r2, x

1000

Interconnection Network

Main Memory

x 1000
The Cache Coherence Problem

P1

```
ld r2, x
add r1, r2, r4
st x, r1
```

P2

```
ld r2, x
```

Interconnection Network

Main Memory

```
x 1000
```

```
2000
```

```
1000
```
The Cache Coherence Problem

ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

P1
2000

add r1, r2, r4
st x, r1

P2
1000

ld r5, x

Interconnection Network

Main Memory

ld r2, x

Should NOT load 1000

ld r5, x

x 1000
Cache Coherence: Whose Responsibility?

- **Software**
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided a cache flush instruction?
    - **FLUSH-LOCAL A**: Flushes/invalidates the cache block containing address A from a processor’s local cache.
    - **FLUSH-GLOBAL A**: Flushes/invalidates the cache block containing address A from all other processors’ caches.
    - **FLUSH-CACHE X**: Flushes/invalidates all blocks in cache X.

- **Hardware**
  - Simplifies software’s job
  - One idea: Invalidate all other copies of block A when a processor writes to it
A Very Simple Coherence Scheme

- Caches “snoop” (observe) each other’s write/read operations. If a processor writes to a block, all others invalidate it from their caches.

- A simple protocol:
  - Write-through, no-write-allocate cache
  - Actions: PrRd, PrWr, BusRd, BusWr
(Non-)Solutions to Cache Coherence

- **No hardware based coherence**
  - Keeping caches coherent is software’s responsibility
    - Makes microarchitect’s life easier
  - Makes average programmer’s life much harder
    - Need to worry about hardware caches to maintain program correctness?
  - Overhead in ensuring coherence in software

- **All caches are shared between all processors**
  - No need for coherence
  - Shared cache becomes the bandwidth bottleneck
  - Very hard to design a scalable system with low-latency cache access this way
Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location.

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order.

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate
  - **Write serialization**: provide a consistent global order seen by all processors

- Need a global point of serialization for this store ordering.
Hardware Cache Coherence

- **Basic idea:**
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy
Coherence: Update vs. Invalidate

- How can we *safely update replicated data*
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy isn’t valid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)
Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

**Update Protocol:**
- Write to block, and simultaneously broadcast written data to sharers
- (Other nodes update their caches if data was present)

**Invalidate Protocol:**
- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes clear block from cache)
Update vs. Invalidate Tradeoffs

Which do we want?
- Write frequency and sharing behavior are critical

**Update**
- If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reenquire (broadcast update pattern)
  - If data is rewritten without intervening reads by other cores, updates were useless
  - Write-through cache policy $\Rightarrow$ bus becomes bottleneck

**Invalidate**
- After invalidation broadcast, core has exclusive access rights
- Only cores that keep reading after each write retain a copy
  - If write contention is high, leads to ping-ponging (rapid mutual invalidation-reenquire)
Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos ISCA 1984]
  - Bus-based, single point of serialization for all requests
  - Processors observe other processors’ actions and infer ownership
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feutrier, 1978]
  - Single point of serialization *per block*, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Directory Based Cache Coherence
Directory Based Coherence

- **Idea:** A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- **An example mechanism:**
  - For each cache block in memory, store $P+1$ bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that the cache that has the only copy of the block and can update it without notifying others
  - On a read: set the cache’s bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache
Directory Based Coherence Example (I)

Example directory based scheme

$P = 4$

| 0 | 0 | 0 | 0 | 0 | 0 |

Exclusive bit

No cache has the block

1. $P_1$ takes a read miss to block A

| 0 | 0 | 0 | 0 | 0 | 0 |

010000

2. $P_3$ takes a read miss

| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

010100
3. P₂ takes a write miss
   → Invalidate P₁ & P₃'s caches
   → Write request → P₂ has the exclusive copy of the block now. Set the Exclusive bit.
   → P₂ can now update the block without notifying any other processor or the directory.
   → P₂ needs to have a bit in its cache indicating it can perform exclusive updates to that block.
   → private/exclusive bit per cache block.

4. P₃ takes a write miss
   → Mem Controller requests block from P₂
   → Mem Controller gives block to P₃
   → P₂ invalidates its copy.

5. P₂ takes a read miss
   → P₃ supplies it.
Snoopy Cache Coherence
Snoopy Cache Coherence

Idea:

- All caches “snoop” all other caches’ read/write requests and keep the cache block coherent
- Each cache block has “coherence metadata” associated with it in the tag store of each cache

Easy to implement if all caches share a common bus

- Each cache broadcasts its read/write operations on the bus
- Good for small-scale multiprocessors
- What if you would like to have a 1000-node multiprocessor?
**SCOPPY CACHE**

Each cache observes its own processor & the bus
- Changes the state of the cached block based on observed actions by processors & the bus

Processor actions to a block:
- **PR** (Proc. Read)
- **RW** (Proc. Write)

Bus actions to a block:
- **BR** (Bus Read)
- **BW** (Bus Write)
  - or **BRx** (Bus Read Exclusive)
Caches “snoop” (observe) each other’s write/read operations

A simple protocol:

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr
A More Sophisticated Protocol: MSI

- Extend single valid bit per block to three states:
  - M (modified): cache line is only copy and is dirty
  - S (shared): cache line is one of several copies
  - I (invalid): not present

- Read miss makes a Read request on bus, saves in S state
- Write miss makes a ReadEx request, saves in M state
- When a processor snoops ReadEx from another writer, it must invalidate its own copy (if any)
- S → M upgrade can be made without re-reading data from memory (via Invl)
More States: MASI

- Invalid → Shared → Modified sequence takes *two* bus ops
- What if data is not shared? Unnecessary broadcasts

- **Exclusive** state: this is the only copy, and it is clean
- Block is *exclusive* if, during *BusRd*, no other cache had it
  - Wired-OR “shared” signal on bus can determine this: snooping caches assert the signal if they also have a copy
- Another *BusRd* also causes transition into *Shared*

- Silent transition **Exclusive** → **Modified** is possible on write!

- MASI is also called the *Illinois protocol* [Papamarcos84]
Illinois Protocol

4 States

M: Modified (Exclusive copy, modified)
E: Exclusive ("","", clean)
S: Shared (Shared copy, clean)
I: Invalid

BI: Invalidate, but already have the data (do not supply it)
BRI: Invalidate, but also need the data (supply it)
MESI State Machine
MESI State Machine

[Culler/Singh96]
MESI State Machine from Lab 7

- **Invalid**
  - Invalidation
  - Cache fill (to multiple)

- **Shared**
  - Upgrade
  - Downgrade
  - 1. Multiple owners (potentially)
  - 2. Read-only access
  - 3. Clean data

- **Modified**
  - Invalidation
  - Downgrade
  - 1. Single owner
  - 2. Read-write access
  - 3. Dirty data

- **Exclusive**
  - Write
  - Downgrade
  - 1. Single owner
  - 2. Read-write access
  - 3. Clean data

Arrows represent state transitions.
MESI State Machine from Lab 7

![MESI State Machine Diagram](image)
Intel Pentium Pro

- Write Allocate
- L1 can have data not in L2
- Hit: Someone has it Clean
- HitM: Someone has it Dirty

Cache Consistency