Lecture 8 Outline

- Memory consistency
  - Sequential consistency
- Invalidation vs. update coherence protocols
- MSI protocol
  - State diagrams
  - Simulation

Switch Gear to Memory Consistency

Warning:
“Studying memory consistency may induce headaches, confusion, apathy, and nausea. Do not try to study it over a short period of time. It is best to study memory consistency over a period of several months by revisiting the problem often, understanding all examples from the lectures and homework exercises.

In addition, it is advised to focus only on Sequential Consistency Model on Chap 5, and leave the rest for Chap 9.”
Let’s Switch Gear to Memory Consistency

Coherence: Writes to a single location are visible to all in the same order
Consistency: Writes to multiple locations are visible to all in the same order
  • Recall Peterson’s algorithm (turn=...; interested[process]=...)
  • When “multiple” = “any”, we have sequential consistency (SC)

Coherence:
Writes to a single location are visible to all in the same order

Consistency:
Writes to multiple locations are visible to all in the same order

• Recall Peterson’s algorithm (turn=...; interested[process]=...)
• When “multiple” = “any”, we have sequential consistency (SC)

P₁
/*Assume initial value of A and ag is 0*/
A = 1;
while (flag == 0); /*spin idly*/
flag = 1;
print A;

P₂

• Sequential consistency (SC) corresponds to our intuition.
• It’s not intuitive to understand memory consistency models!
• Coherence doesn’t help; it pertains only to a single location

Another Example of Ordering

P₁
/*Assume initial values of A and B are 0*/
(1a) A = 1;
(1b) B = 2;

P₂
/*Assume initial values of A and B are 0*/
(2a) print B;
(2b) print A;

• What do you think the results should be? You may think:
  • 1a, 1b, 2a, 2b ⇒ {A=1, B=2} (programmers’ intuition:
  • 1a, 2a, 2b, 1b ⇒ {A=1, B=0} sequential consistency
  • 2a, 2b, 1a, 1b ⇒ {A=0, B=0}
  • Is {A=0, B=2} possible? Yes, suppose P2 sees: 1b, 2a, 2b, 1a e.g. evil compiler, evil interconnection

• Intuition? Whatever it is, need an ordering model for clear semantics across different locations as well (vs. just cache coherence!) so programmers can reason about what results are possible.
A Memory-Consistency Model …

- Is a contract between programmer and system
  - Necessary to reason about correctness of shared-memory programs

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it

- Implications for programmers
  - Restricts algorithms that can be used
  - e.g., Peterson’s algorithm, home-brew synchronization will be incorrect in machines that do not guarantee SC

- Implications for compiler writers and arch designers
  - Determine how much accesses can be reordered

Lecture 8 Outline

- Memory consistency
  - Sequential consistency

- Invalidation vs. update coherence protocols

- MSI protocol
  - State diagrams
  - Simulation
Sequential Consistency

“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.” [Lamport, 1979]

- (as if there were no caches, and a single memory)
- Total order achieved by interleaving accesses from different processes
- Maintains program order, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others

What Really Is Program Order?

- Intuitively, the order in which operations appear in source code
- Thus, we assume order as seen by programmer,
  - the compiler is prohibited from reordering mem accesses to shared variables.

- Note that this is one reason parallel programs are less efficient than serial programs.
What Reordering Is Safe in SC?

What matters is order in which it appears to execute, not order in which it executes.

\[
\begin{array}{c|c}
\text{P}_1 & \text{P}_2 \\
\hline
\text{/*Assume initial values of A and B are 0*/} \\
(1a) & A = 1; \\
(1b) & B = 2; \\
(2a) & \text{print } B; \\
(2b) & \text{print } A; \\
\end{array}
\]

- possible outcomes for (A,B): (0,0), (1,0), (1,2); impossible under SC: (0,2)
- we know 1a → 1b and 2a → 2b by program order
- A = 0 implies 2b → 1a, which implies 2a→1b
- B = 2 implies 1b → 2a, which leads to a contradiction

BUT, actual execution 1b → 1a → 2b → 2a is SC, despite not program order
  - appears just like 1a → 1b → 2a → 2b as visible from results
  - actual execution 1b → 2a → 2b → is not SC
  - Thus, some reordering is possible, but difficult to reason that it ensures SC

Conditions for SC

- Two kinds of requirements
  - **Program order**
    - memory operations issued by a process must appear to become visible (to others and itself) in program order
  - **Global Order**
    - Atomicity: one memory operation should appear to complete with respect to all processes before the next one is issued
    - Global order = operation order is consistent as seen by all processes
- Tricky part: how to make writes atomic?
  - Detect write completion
  - Read completion is easy: it completes when the data returns
- Who should enforce SC?
  - Compiler should not change program order
  - Hardware should ensure program order and atomicity
Write Atomicity

- **Write Atomicity**: ensures write ordering same for all processes
  - In effect, extends write serialization to writes from multiple processes

  ![Diagram of processes](image)

  - Transitivity implies A should print as 1 under SC
  - Problem if
    - $P_2$ leaves loop, writes B, and $P_3$ sees new B but old A (from its cache, say)

Pitfalls for SC

- **Compiler optimizations**
  - Loop transformations: Reorder loads and stores
  - Register allocation: Actually eliminates some loads/stores
    - Declare the variables as “volatile” to disallow this.

- **Hardware may violate SC for better performance**
  - Write buffers
  - Out-of-order execution
  - Interconnection-network delay
Is the Write-Through Example SC?

- Assume no write buffers, load-store bypassing
- Yes, because of the atomic bus:
  - Any write and read misses (to all locations) serialized by bus into bus order
  - If read obtains value of write W, W guaranteed to have completed
    - since it caused a bus transaction
  - When write W is performed w.r.t. any processor, all previous writes in bus order have completed

Lecture 8 Outline

- Memory consistency
  - Sequential consistency
- Invalidation vs. update coherence protocols
- MSI protocol
  - State diagrams
  - Simulation
Using Write-Back Caches

- **Dirty state**
  - Uniprocessor: Line has been modified
  - Multiprocessor: Line has been modified + exclusive ownership
    - **Exclusive**: “I’m the only one that have it, other than possibly the main memory.”
    - I’m the **Owner**: responsible for supplying block upon a request for it.

- **Two variations**
  - Invalidation based protocol
  - Update-based protocols

Invalidation-Based Protocols

- **Idea**: On my write, invalidate everybody else \(\rightarrow\) I get exclusive state.
- **Exclusive**
  - Can modify without notifying anyone else (i.e., without bus transaction)
  - Must first get block in exclusive state before writing into it
  - Even if already in valid state, need bus transaction to invalidate (Read Exclusive = RdX)
- **Read and Read-exclusive bus transactions drive coherence actions**
- **On writeback**: Silent replace if not in D (or M=Modified), otherwise flush/write-back \(???)\
Update-Based Protocols

- Idea: On my write, update everybody else (with new word)
  - New bus transaction: Update
- Advantages
  - Other processors don’t miss on next access
  - Saves refetch: In invalidation protocols, they would miss & bus transaction.
  - Saves bandwidth: Single bus transaction updates several caches
- Disadvantages
  - Multiple writes by same processor cause multiple update transactions
    - In invalidation, first write gets exclusive ownership, others local
  - Detailed tradeoffs more complex

Invalidate versus Update

- Is a block written by one processor read by others before it is rewritten?
- Invalidation:
  - Yes → Readers will take a miss
  - No → Multiple writes without additional traffic
    - and clears out copies that won’t be used again
- Update:
  - Yes → Readers will not miss if they had a copy previously
    - single bus transaction to update all copies
  - No → Multiple useless updates, even to dead copies
- Invalidation protocols much more popular (more later)
  - Some systems provide both, or even hybrid
Lecture 8 Outline

- Memory consistency
  - Sequential consistency
- Invalidation vs. update coherence protocols
- MSI protocol
  - State diagrams
  - Simulation

Basic MSI Writeback Inval Protocol

- States
  - Invalid (I)
  - Shared (S): one or more copies, and memory copy is up-to-date
  - Dirty or Modified (M): only one copy
- Processor Events:
  - PrRd (read), PrWr (write)
- Bus Transactions
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify (instead of BusWr)
  - Flush: updates memory
- Actions
  - Update state, perform bus transaction, flush value onto bus
State-Transition Diagrams

- On the following slides, we will display the state-transition diagrams
  - for processor-initiated transactions
  - for bus-initiated transactions
- We will see transitions of the following form:
  - Invalidation: \( \langle \text{Any} \rangle \rightarrow I \)
  - Intervention: \( \langle \text{Exclusive, Modified} \rangle \rightarrow \text{Shared} \)

---

MSI: Processor-Initiated Transactions
MSI: Bus-Initiated Transactions

Thus, valid data must be supplied by memory.

MSI State Transition Diagram
MSI Visualization

MSI Visualization
MSI Visualization

P1

X=2

Mem Ctrl

X=1

P2

Snooper

Snooper

P3

rd &X

BusRd

Mem Ctrl

X=2

Flush

Cancel memory read

ECE/CSC 506 - Summer 2006 - E. F. Gehringer, based on slides by Yan Solihin
MSI Visualization

MSI Visualization

Update memory as well
MSI Visualization

![Diagram of MSI Visualization](image)

Lecture 8
ECE/CSC 506 - Summer 2006 - E. F. Gehringer, based on slides by Yan Solihin

34
Example: Rd/Wr to a single line

<table>
<thead>
<tr>
<th>Proc Action</th>
<th>State P1</th>
<th>State P2</th>
<th>State P3</th>
<th>Bus Action</th>
<th>Data From</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>S</td>
<td>-</td>
<td>-</td>
<td>BusRd</td>
<td>Mem</td>
</tr>
<tr>
<td>W1</td>
<td>M</td>
<td>-</td>
<td>-</td>
<td>BusRdX*</td>
<td>Mem</td>
</tr>
<tr>
<td>R3</td>
<td>S</td>
<td>-</td>
<td>S</td>
<td>BusRd</td>
<td>P1 cache</td>
</tr>
<tr>
<td>W3</td>
<td>I</td>
<td>-</td>
<td>M</td>
<td>BusRdX*</td>
<td>Mem</td>
</tr>
<tr>
<td>R1</td>
<td>S</td>
<td>-</td>
<td>S</td>
<td>BusRd</td>
<td>P3 cache</td>
</tr>
<tr>
<td>R3</td>
<td>S</td>
<td>-</td>
<td>S</td>
<td>-</td>
<td>Local Cache</td>
</tr>
<tr>
<td>R2</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>BusRd</td>
<td>Mem</td>
</tr>
</tbody>
</table>

*or, BusUpgr (data from own cache)

Notes on MSI Protocol

- For M → I, BusRdX/Flush: why flush? Because it is a read with intention to write, as opposed to write.
  - Thus, there is a possibility for a read before the write is performed
  - In addition, the writes could be to different words in a line
- Write to shared block:
  - Already have latest data; can use upgrade (BusUpgr) instead of BusRdX
- Replacement changes state of two blocks: outgoing and incoming
- Flush has to modify both caches and main memory

Note: coherence granularity is u (a single line). What happens when all the reads go to word 0 on line u, but write by P3 goes to word 1 on line u? False-sharing miss on the 2nd R1
MSI: Coherence and SC

- **Coherence:**
  - Write propagation:
    - through invalidation, and flush on subsequent BusRds
  - Write serialization?
    - Writes (BusRdX) that go to the bus appear in bus order (and handled by snoopsers in bus order!)
    - Writes that do not go to the bus?
      - Only happens the line state is M, i.e. when I am the only processor holding the line. Local writes are only visible to me, so they are serialized.

- **To enforce SC:**
  - Program order: enforced by following the bus transaction order
    - All writes appear on the bus
    - All local writes (within 1 processor) can follow program order
  - Write completion: Occurs when write appears on bus
  - Write atomicity: A read returns the latest value of a write. At that time, the value is visible to all others (on a bus transaction, or on a local write).