Lecture 26: Multiprocessors

• Today’s topics:
  ▪ Directory-based coherence
  ▪ Synchronization
  ▪ Consistency
  ▪ Shared memory vs message-passing
Cache Coherence Protocols

• Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

• Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  ➢ Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  ➢ Write-update: when a processor writes, it updates other shared copies of that block
Coherence in Distributed Memory Multiprocs

• Distributed memory systems are typically larger → bus-based snooping may not work well

• Option 1: software-based mechanisms – message-passing systems or software-controlled cache coherence

• Option 2: hardware-based mechanisms – directory-based cache coherence
Distributed Memory Multiprocessors

Interconnection network
Directory-Based Cache Coherence

• The physical memory is distributed among all processors.

• The directory is also distributed along with the corresponding memory.

• The physical address is enough to determine the location of memory.

• The (many) processing nodes are connected with a scalable interconnect (not a bus) – hence, messages are no longer broadcast, but routed from sender to receiver – since the processing nodes can no longer snoop, the directory keeps track of sharing state.
Cache Block States

• What are the different states a block of memory can have within the directory?

• Note that we need information for each cache so that invalidate messages can be sent

• The directory now serves as the arbitrator: if multiple write attempts happen simultaneously, the directory determines the ordering
Directory-Based Example

A: Rd X
B: Rd X
C: Rd X
A: Wr X
A: Wr X
C: Wr X
B: Rd X
A: Rd X
A: Rd Y
B: Wr X
B: Rd Y
B: Wr X
B: Wr Y

Interconnection network
## Example

<table>
<thead>
<tr>
<th>Request</th>
<th>Cache Hit/Miss</th>
<th>Messages</th>
<th>Dir State</th>
<th>State in C1</th>
<th>State in C2</th>
<th>State in C3</th>
<th>State in C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Rd X Miss</td>
<td>Rd-req to Dir. Dir responds.</td>
<td>X: S: 1</td>
<td>S</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Rd X Miss</td>
<td>Rd-req to Dir. Dir responds.</td>
<td>X: S: 1, 2</td>
<td>S</td>
<td>S</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Wr X Perms Miss</td>
<td>Upgr-req to Dir. Dir sends INV to P1. P1 sends ACK to Dir. Dir grants perms to P2.</td>
<td>X: M: 2</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P3: Wr X Write Miss</td>
<td>Wr-req to Dir. Dir fwds request to P2. P2 sends data to Dir. Dir sends data to P3.</td>
<td>X: M: 3</td>
<td>Inv</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P3: Rd X Read Hit</td>
<td>-</td>
<td>Inv</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P4: Rd X Read Miss</td>
<td>Rd-req to Dir. Dir fwds request to P3. P3 sends data to Dir. Memory wrtbk. Dir sends data to P4.</td>
<td>X: S: 3, 4</td>
<td>Inv</td>
<td>Inv</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>
Directory Actions

• If block is in uncached state:
  ➢ Read miss: send data, make block shared
  ➢ Write miss: send data, make block exclusive

• If block is in shared state:
  ➢ Read miss: send data, add node to sharers list
  ➢ Write miss: send data, invalidate sharers, make excl

• If block is in exclusive state:
  ➢ Read miss: ask owner for data, write to memory, send data, make shared, add node to sharers list
  ➢ Data write back: write to memory, make uncached
  ➢ Write miss: ask owner for data, write to memory, send data, update identity of new owner, remain exclusive
Constructing Locks

• Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

• A lock surrounding the data/code ensures that only one program can be in a critical section at a time

• The hardware must provide some basic primitives that allow us to construct locks with different properties

Parallel (unlocked) banking transactions

Bank balance $1000
Rd $1000
Add $100
Wr $1100
Rd $1000
Add $200
Wr $1200
Synchronization

• The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

• Atomic exchange: swap contents of register and memory

• Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory (if memory has 0, lock is free)

• lock:  t&s  register, location
         bnz  register, lock
         CS
         st  location, #0

When multiple parallel threads execute this code, only one will be able to enter CS
Coherence Vs. Consistency

• Recall that coherence guarantees (i) write propagation (a write will eventually be seen by other processors), and (ii) write serialization (all processors see writes to the same location in the same order)

• The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions
Consistency Example

• Consider a multiprocessor with bus-based snooping cache coherence

<table>
<thead>
<tr>
<th>Initially A = B = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1  P2</td>
</tr>
<tr>
<td>A ← 1  B ← 1</td>
</tr>
<tr>
<td>... ...</td>
</tr>
<tr>
<td>if (B == 0)  if (A == 0)</td>
</tr>
<tr>
<td>Crit.Section  Crit.Section</td>
</tr>
</tbody>
</table>
Consistency Example

• Consider a multiprocessor with bus-based snooping cache coherence

Initially $A = B = 0$

\[
\begin{array}{ll}
P1 & P2 \\
A & B \\
\leftarrow 1 & \leftarrow 1 \\
\ldots & \ldots \\
\text{if } (B == 0) & \text{if } (A == 0) \\
\text{Crit. Section} & \text{Crit. Section}
\end{array}
\]

The programmer expected the above code to implement a lock – because of ooo, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware’s reordering capabilities
Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achieveable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion.

- The multiprocessor in the previous example is not sequentially consistent.

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow.
Shared-Memory Vs. Message-Passing

**Shared-memory:**
- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

**Message-passing:**
- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Software-controlled caching
- Sender can initiate data transfer
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
    diff = 0;
    for i ← 1 to n do
      for j ← 1 to n do
        temp = A[i,j];
        A[i,j] ← 0.2 * (A[i,j] + neighbors);
        diff += abs(A[i,j] – temp);
      end for
    end for
    if (diff < TOL) then done = 1;
  end while
end procedure
Shared Address Space Model

int n, nprocs;
float **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
read(n); read(nprocs);
A ← G_MALLOC();
initialize (A);
CREATE (nprocs,Solve,A);
WAIT_FOR_END (nprocs);
end main

procedure Solve(A)
int i, j, pid, done=0;
float temp, mydiff=0;
int mymin = 1 + (pid * n/nprocs);
int mymax = mymin + n/nprocs -1;
while (!done) do
  mydiff = diff = 0;
  BARRIER(bar1,nprocs);
  for i ← mymin to mymax
    for j ← 1 to n do
      ...
    endfor
  endfor
  LOCK(diff_lock);
  diff += mydiff;
  UNLOCK(diff_lock);
  BARRIER (bar1, nprocs);
  if (diff < TOL) then done = 1;
  BARRIER (bar1, nprocs);
endwhile
Message Passing Model

main()
read(n); read(nprocs);
CREATE (nprocs-1, Solve);
Solve();
WAIT_FOR_END (nprocs-1);

procedure Solve()
int i, j, pid, nn = n/nprocs, done=0;
float temp, tempdiff, mydiff = 0;
myA \leftarrow \text{malloc}(\ldots)
initialize(myA);
while (!done) do
    mydiff = 0;
    if (pid != 0)
        SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
        SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
        RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
        RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i \leftarrow 1 to nn do
        for j \leftarrow 1 to n do
            \ldots
        endfor
    endfor
    if (pid != 0)
        SEND(mydiff, 1, 0, DIFF);
    RECEIVE(done, 1, 0, DONE);
    else
        for i \leftarrow 1 to nprocs-1 do
            RECEIVE(tempdiff, 1, *, DIFF);
        endfor
        mydiff += tempdiff;
    endif
    if (mydiff < TOL)  done = 1;
    for i \leftarrow 1 to nprocs-1 do
        SEND(done, 1, I, DONE);
    endfor
endwhile
Title

• Bullet