Lecture: Coherence Protocols

- Topics: multi-thread programming models, snooping-based protocols, directory-based protocols
Multiprocs -- Memory Organization - I

• Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)

• Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)

• Shared-memory because all processors can access the entire memory address space

• Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors
SMPs or Centralized Shared-Memory

Diagram:
- Processor
  - Caches
- Processor
  - Caches
- Processor
  - Caches
- Processor
  - Caches
- Main Memory
- I/O System
Multiprocs -- Memory Organization - II

- For higher scalability, memory is distributed among processors → distributed memory multiprocessors

- If one processor can directly address the memory local to another processor, the address space is shared → distributed shared-memory (DSM) multiprocessor

- If memories are strictly local, we need messages to communicate data → cluster of computers or multicomputers

- Non-uniform memory architecture (NUMA) since local memory has lower latency than remote memory
SMPs

- Centralized main memory and many caches → many copies of the same data

- A system is cache coherent if a read returns the most recently written value for that word

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Value of X in Cache-A</th>
<th>Cache-B</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU-A reads X</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU-B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU-A stores 0 in X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
A memory system is coherent if:

- Write propagation: P1 writes to X, sufficient time elapses, P2 reads X and gets the value written by P1
- Write serialization: Two writes to the same location by two processors are seen in the same order by all processors
- The memory \textit{consistency} model defines “time elapsed” before the effect of a processor is seen by others and the ordering with R/W to other locations (loosely speaking – more later)
SMPs or Centralized Shared-Memory
Design Issues

- Invalidate
- Find data
- Writeback / writethrough

- Cache block states
- Contention for tags
- Enforcing write serialization
SMP Example

Processor A
Caches

Processor B
Caches

Processor C
Caches

Processor D
Caches

Main Memory

I/O System

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

• P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
• P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1 does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state

• P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
• P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated
## Example

<table>
<thead>
<tr>
<th>Request</th>
<th>Cache Hit/Miss</th>
<th>Request on the bus</th>
<th>Who responds</th>
<th>State in Cache 1</th>
<th>State in Cache 2</th>
<th>State in Cache 3</th>
<th>State in Cache 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Rd X</td>
<td>Miss</td>
<td>Rd X</td>
<td>Memory</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Rd X</td>
<td>Miss</td>
<td>Rd X</td>
<td>Memory</td>
<td>S</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Wr X</td>
<td>Perms Miss</td>
<td>Upgrade X</td>
<td>No response. Other caches invalidate.</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P3: Wr X</td>
<td>Write Miss</td>
<td>Wr X</td>
<td>P2 responds</td>
<td>Inv</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>P3: Rd X</td>
<td>Read Hit</td>
<td>-</td>
<td></td>
<td>Inv</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>P4: Rd X</td>
<td>Read Miss</td>
<td>Rd X</td>
<td>P3 responds. Mem wrtbk</td>
<td>Inv</td>
<td>Inv</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>
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


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.
Distributed Memory Multiprocessors
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
B: Wr X
B: Wr Y
B: Wr Y
## 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</td>
<td>Miss</td>
<td>Ld-req to Dir. Dir responds.</td>
<td>X: S: 1</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Rd X</td>
<td>Miss</td>
<td>Ld-req to Dir. Dir responds.</td>
<td>X: S: 1, 2</td>
<td>S</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
<tr>
<td>P2: Wr X</td>
<td>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>
</tr>
<tr>
<td>P3: Wr X</td>
<td>Write Miss</td>
<td>Wr-req to Dir. Dir fwd 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>
</tr>
<tr>
<td>P3: Rd X</td>
<td>Read Hit</td>
<td>-</td>
<td>-</td>
<td>Inv</td>
<td>Inv</td>
<td>M</td>
<td>Inv</td>
</tr>
<tr>
<td>P4: Rd X</td>
<td>Read Miss</td>
<td>Ld-req to Dir. Dir fwd 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>
</tr>
</tbody>
</table>
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 block state is also stored in the cache for efficiency

• The directory now serves as the arbitrator: if multiple write attempts happen simultaneously, the directory determines the ordering
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
• Sender can initiate data transfer
Ocean Kernel

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
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/procs);
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}(...)
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
            ...
        endfor
    endfor
    if (mydiff < TOL) done = 1;
    for i \leftarrow 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
    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);
            mydiff += tempdiff;
        endfor
        if (mydiff < TOL) done = 1;
        for i \leftarrow 1 to nprocs-1 do
            SEND(done, 1, I, DONE);
        endfor
    endif
endwhile