Lecture 3: Coherence Protocols

- Topics: message-passing programming model, coherence protocol examples

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
```

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/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 ← 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 ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if  (mydiff < TOL)  done = 1;
      for i ← 1 to nprocs-1  do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

Message Passing Model

- Note that each process has two additional rows to store data produced by its neighbors
- Unlike the shared memory model, execution is deterministic -- two executions will produce the same result
- A send-receive match is a synchronization event – hence, we no longer need locks while updating the diff counter, or barriers while allowing processes to proceed with the next iteration

Models for SEND and RECEIVE

- Synchronous: SEND returns control back to the program only when the RECEIVE has completed – will it work for the program on the previous slide?

- Blocking Asynchronous: SEND returns control back to the program after the OS has copied the message into its space -- the program can now modify the sent data structure

- Nonblocking Asynchronous: SEND and RECEIVE return control immediately – the message will get copied at some point, so the process must overlap some other computation with the communication – other primitives are used to probe if the communication has finished or not

Cache Coherence

A multiprocessor system is cache coherent if

- a value written by a processor is eventually visible to reads by other processors – write propagation

- two writes to the same location by two processors are seen in the same order by all processors – write serialization

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

Protocol-I MSI

- 3-state write-back invalidation bus-based snooping protocol
- Each block can be in one of three states – invalid, shared, modified (exclusive)
- A processor must acquire the block in exclusive state in order to write to it – this is done by placing an exclusive read request on the bus – every other cached copy is invalidated
- When some other processor tries to read an exclusive block, the block is demoted to shared

Design Issues, Optimizations

- When does memory get updated?
  - demotion from modified to shared?
  - move from modified in one cache to modified in another?

- Who responds with data? – memory or a cache that has the block in exclusive state – does it help if sharers respond?

- We can assume that bus, memory, and cache state transactions are atomic – if not, we will need more states

- A transition from shared to modified only requires an upgrade request and no transfer of data

- Is the protocol simpler for a write-through cache?

4-State Protocol

- Multiprocessors execute many single-threaded programs
- A read followed by a write will generate bus transactions to acquire the block in exclusive state even though there are no sharers
- Note that we can optimize protocols by adding more states – increases design/verification complexity

MESI Protocol

- The new state is exclusive-clean – the cache can service read requests and no other cache has the same block
- When the processor attempts a write, the block is upgraded to exclusive-modified without generating a bus transaction
- When a processor makes a read request, it must detect if it has the only cached copy – the interconnect must include an additional signal that is asserted by each cache if it has a valid copy of the block

Design Issues

- When caches evict blocks, they do not inform other caches – it is possible to have a block in shared state even though it is an exclusive-clean copy

- Cache-to-cache sharing: SRAM vs. DRAM latencies, contention in remote caches, protocol complexities (memory has to wait, which cache responds), can be especially useful in distributed memory systems

- The protocol can be improved by adding a fifth state (owner – MOESI) – the owner services reads (instead of memory)

Update Protocol (Dragon)

- 4-state write-back update protocol, first used in the Dragon multiprocessor (1984)
- Write-back update is not the same as write-through – on a write, only caches are updated, not memory
- Goal: writes may usually not be on the critical path, but subsequent reads may be

4 States

- No invalid state
- Modified and Exclusive-clean as before: used when there is a sole cached copy
- Shared-clean: potentially multiple caches have this block and main memory may or may not be up-to-date
- Shared-modified: potentially multiple caches have this block, main memory is not up-to-date, and this cache must update memory – only one block can be in Sm state
- In reality, one state would have sufficed – more states to reduce traffic

Design Issues

- If the update is also sent to main memory, the Sm state can be eliminated
- If all caches are informed when a block is evicted, the block can be moved from shared to M or E – this can help save future bus transactions
- Having an extra wire to determine exclusivity seems like a worthy trade-off in update systems

Examples

	P1			P2		
	MSI	MESI	Dragon	MSI	MESI	Dragon
• P1: Rd X						
• P1: Wr X						
• P2: Rd X						
• P1: Wr X						
• P1: Wr X						
• P2: Rd X						
• P2: Wr X						
Total transfers:						

Title

- Bullet