G22.2243-001
High Performance Computer Architecture

Lecture 10
Multiprocessing (Cont’d)

November 7, 2007

Outline

- Announcements
  - HW Assignment 3 due back today
  - Lab Assignment 3 due in a week: Nov 14
- Multiprocessors
  - Coherence protocols
    - Snooping-based protocols (review)
    - Directory-based protocols

Directory Protocol (Cont’d)

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  - P = processor number, A = address

Directory Protocol Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>Processor P has a read miss at address A; request data and make P a read sharer</td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>Processor P has a write miss at address A; request data and make P exclusive owner</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>Invalidate a shared copy of data at address A</td>
</tr>
<tr>
<td>Fetch</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>Fetch block at address A &amp; send it to its home directory; change state to shared at remote</td>
</tr>
<tr>
<td>Fetch/Invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>Fetch block at address A &amp; send it to its home directory; invalidate the block in the cache</td>
</tr>
<tr>
<td>Data value reply</td>
<td>Home directory</td>
<td>Local cache</td>
<td>Return a data value from the home memory</td>
</tr>
<tr>
<td>Data write-back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>Write-back a data value for address A</td>
</tr>
</tbody>
</table>

CPU - Cache State Machine

- State machine for each Cache block
- Invalid state if in memory

State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snooping case
- Transactions very similar
- Transitions caused by read misses, write misses, invalidates, and data fetch requests
- Generates read miss & write miss messages to home directory
- Write misses that were broadcast on the bus for snooping
  - explicit invalidate & data fetch requests

State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- Two actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message

Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory & requestor made (the first) sharing node; state of block made Shared
  - Write miss: requesting processor is sent the value. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared => the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages. Sharers is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol (Cont’d)

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner).
- Three possible directory requests:
  - Read miss: owner processor is sent a data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor
  Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy); state is shared
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date
    (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty
  - Write miss: block has a new owner. A message is sent to old owner (fetch/invalidate) causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive. The old owner’s cache block status becomes Invalid

Directory State Machine

- State machine for Directory requests for each memory block
- Uncached state if in memory

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block

Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network
- Optimation:
  - read miss or write miss in Exclusive: send data directly to requestor from owner vs. first to memory and then from memory to requestor

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptible instruction to fetch and update memory (atomic operation)
  - User level synchronization operation using this primitive
  - For large scale MP's, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction: Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
  0 => synchronization variable is free
  1 => synchronization variable is locked and unavailable
  - Set registry to 1 & swap
  - New value in registry determines success in getting lock
  0 if you succeeded in setting the lock (you were first)
  1 if other processor had already claimed access
  - Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  0 => synchronization variable is free

Uninterruptable Instruction: Fetch and Update Memory (Cont'd)

- Hard to have read & write in 1 instruction: use 2 instead
  - Load linked (or load locked) + store conditional
    - Load linked returns the initial value
    - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- Example doing atomic swap with LL & SC:
  try: mov R3,R4 ; mov exchange value
  ll R2,0(R1) ; load linked
  sc R2,0(R1) ; store conditional
  beqz R3,try  ; branch store fails (R3 = 0)
mov R4,R2  ; put load value in R4

- Example doing fetch & increment with LL & SC:
  try: ll R2,0(R1) ; load linked
  add R2,0(R1) ; increment (OK if reg-reg)
sr R2,0(R1) ; store conditional
  beqz R2,try ; branch store fails (R2 = 0)
User Level Synchronization

• Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
  ```
  li R2,#1
  lockit: beq R2,(R1) ;atomic exchange
  bnez R2,lockit ;already locked?
  ```
• What about MP with cache coherency?
  – Want to spin on cache copy to avoid full memory latency
  – Likely to get cache hits for such variables
• Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
• Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):
  ```
  try: li R2,#1
  lockit: lw R3,0(R1) ;load var
  bnez R3,lockit ;not free=>spin
  exch R2,0(R1) ;atomic exchange
  bnez R2,try ;already locked?
  ```

Another MP Issue:
Memory Consistency Models

• What is consistency? When must a processor see the new value? e.g.,
  seems that
  ```
P1: A = 0
  A = 1
  L1: if (B == 0) ...
  L2: if (A == 0) ...
  ```
• Impossible for both if statements L1 & L2 to be true?
  – What if write invalidate is delayed & processor continues?
• Memory consistency models: what are the rules for such cases?
  • Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
    – SC: delay all memory accesses until all invalidates done

Memory Consistency Model

• Schemes faster execution to sequential consistency
• Not really an issue for most programs; they are synchronized
  – A program is synchronized if all access to shared data are ordered by synchronization operations
    – A program is synchronized if all access to shared data are ordered by synchronization operations
    – write (s)
    – release (s) [unlock]
    – acquire (s) [lock]
    – read(s)
• Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(processor speed)
• Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Summary

• Caches contain all information on state of cached memory blocks
• Snooping and Directory Protocols similar
• Bus makes snooping easier because of broadcast
  – Uniform Memory Access
• Directory has extra data structure to keep track of state of all memory blocks
  – Distributing directory
  – Scalable shared address multiprocessor
  – Non Uniform Memory Access (NUMA)

SGI Origin 2000

• a pure NUMA
• 2 CPUs per node,
• Scales up to 2048 processors
• Design for scientific computation vs. commercial processing
• Scalable bandwidth is crucial to Origin
• Show average cycles per memory reference in 4 categories:
  – Cache Hit
  – Miss to local memory
  – Remote miss to home
  – 3-network hop miss to remote cache

Scientific/Technical Parallel Apps

• FFT Kernel: 1D complex number FFT
  – 2 matrix transpose phases => all-to-all communication
  – Sequential time for n data points: O(n log n)
  – Example is 1 million point data set
• LU Kernel: dense matrix factorization
  – Blocking helps cache miss rate, 16x16
  – Sequential time for nxn matrix: O(n^3)
  – Example is 512 x 512 matrix
Scientific/Technical Parallel Apps (Cont’d)

• Barnes App: Barnes-Hut n-body algorithm solving a problem in galaxy evolution
  – n-body algorithms rely on forces drop off with distance
  – if far enough away, can ignore (e.g., gravity is 1/d²)
  – Sequential time for n data points: O(n log n)
  – Example is 16,384 bodies

• Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations
  – red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors
  – Multigrid solve finite differential equations by iteration using hierarchical grids
  – Communication when boundary accessed by adjacent subgrid
  – Sequential time for n x n grid: O(n²)
  – Input: 130 x 130 grid points, 5 iterations

Cross Cutting Issues: Performance Measurement of Parallel Processors

• Performance: how well scale as number of processors increases
  • Speedup fixed as well as scaleup of problem
    – Assume benchmark of size n on p processors makes sense: how scale benchmark to run on m * p processors?
    – Memory-constrained scaling: keeping the amount of memory used per processor constant
    – Time-constrained scaling: keeping total execution time, assuming perfect speedup, constant
  • Example: 1 hour on 10 P, time ~ O(n³), 100 P?
    – Time-constrained scaling: 1 hour, => 10³/³n => 10³ or 100X or 100 hours! 10X processors for 100X longer???
    – Need to know application well to scale: # iterations, error tolerance

Cross Cutting Issues: Memory System Issues

• Multi-level cache hierarchy + multi-level inclusion—every level of cache hierarchy is a subset of next level—then can reduce contention between coherence traffic and processor traffic
  – Hard if cache blocks different sizes
  – Also issues in memory consistency model and speculation, nonblocking caches, prefetching

FFT and LU Kernels

• 512 KB two-way data cache with 64-byte blocks

Cross Cutting Issues: Measuring MP performance by linear speedup v. execution time

• “linear speedup” graph of performance as scale CPUs
  • Compare best algorithm on each computer
  • Relative speedup - run same program on MP and uniprocessor
    – But parallel program may be slower on a uniprocessor than a sequential version
    – Or developing a parallel program will sometimes lead to algorithmic improvements, which should also benefit uni
  • True speedup - run best program on each machine
  • Can get superlinear speedup due to larger effective cache with more CPUs
Fallacy: Linear speedups are needed to make multiprocessors cost-effective

- Mark Hill & David Wood 1995 study
- Compare costs SGI uniprocessor and MP
  - Uniprocessor = $38,400 + $100 * MB
  - MP = $81,600 + $20,000 * P + $100 * MB
- What speedup for better MP cost performance?
  - 8 proc $344k; $344k/138k = 2.5X
  - 16 proc need only 3.6X, or 25% linear speedup
- Even if need some more memory for MP, not linear

Fallacy: Multiprocessors are “free”

- “Since microprocessors contain support for snooping caches, can build small-scale, bus-based multiprocessors for no additional cost”
- Need more complex memory controller (coherence) than for uniprocessor
- Memory access time always longer with more complex controller
- Additional software effort: compilers, operating systems, and debuggers all must be adapted for a parallel system

Multiprocessors

- Some optimism about future
  - Parallel processing beginning to be understood in some domains
  - More performance than that achieved with a single-chip microprocessor
  - MPs are highly effective for multiprogrammed workloads
  - MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DNS applications (where query optimization is critical), and large-scale, web searching applications
  - On-chip MPs appears to be growing
    1) embedded market where natural parallelism often exists an obvious alternative to faster less silicon efficient, CPU
    2) diminishing returns in high-end microprocessor encourage designers to pursue on-chip multiprocessing

Lab Assignment 3

A Quick Look

- Simulator sim-multfu:
  - In-order issue
  - Out-of-order execution
  - In-order commit
- Pipeline stages:
  - Fetch (IF)
  - Dispatch (ID1)
  - Issue (ID2)
    - EXE (2nd cycle)
    - EXE (3rd cycle)
    - ...
    - Writeback (WB)
    - Commit (CM)

Multi-cycle Functional Units (FUs)

- Multi cycle functional units (FUs)
  - Operation latency
    - Cycles until result is ready for use (ready for WB in “current cycle + operation latency” cycle)
  - Issue latency
    - number of cycles before another operation can be issued on this resource (ready for issue in “current cycle + issue latency” cycle)
- char *name; /* name of functional unit */
- int quantity; /* total instances of this unit */
- int busy; /* non-zero if this unit is busy */
- int class; /* matching resource class */
- int opplat; /* operation latency: cycles until result is ready for use */
- int issuelat; /* issue latency: number of cycles before another operation can be issued on this resource */
- "integer-ALU", 1, 0, {{IntALU, 1, 1}}
- "integer-MULT/DIV", 1, 0, {{IntMULT, 3, 1}, {IntDIV, 20, 19}}
- "memory-port", 1, 0, {{RdPort, 1, 1}, {WrPort, 1, 1}}
- "FP-adder", 1, 0, {{FloatADD, 2, 1}, {FloatCMP, 2, 1}, {FloatCVT, 2, 1}}
- "FP-MULT/DIV", 1, 0, {{FloatMULT, 4, 1}, {FloatDIV, 12, 12}, {FloatSQRT, 24, 24}}
Reservation Stations

struct reservation_station {
    /* instruction info */
    md_inst_t IR;
    enum md_opcode op;
    md_addr_t PC;
    md_addr_t addr; /* effective address for LD/STs */
    int will_exit;
    /* RS info */
    INST_TAG_TYPE tag;  /* reservation station tag: increment to invalidate RS */
    INST_SEQ_TYPE seq;  /* instruction sequence, used to sort the ready list and tag instruction */
    int in_LSQ; /* non-zero if op is in LSQ */
    int ea_comp; /* non-zero if op is an addr comp */
    /* instruction status */
    int queued; /* operands ready and queued */
    int issued; /* operation is/was executing */
    int completed; /* operation has completed execution */
    /* output dependent links */
    int onames[MAX_ODEPS]; /* output logical names */
    int odep_ready[MAX_ODEPS]; /* output operand ready? */
    /* input dependent links */
    int inames[MAX_IDEPS]; /* input logical names */
    int idep_ready[MAX_IDEPS]; /* input operand ready? */
};

Circular Queues

/* Register Update Unit (RUU): combination of reservation stations and reorder buffer device, organized as a circular queue */
static struct reservation_station *RUU; /* register update unit */
static int RUU_head, RUU_tail; /* RUU head and tail */
static int RUU_num; /* num entries in RUU */

/* Load/Store Queue (LSQ): holds loads and stores in program order, indicating status of load/store accesses (will be used for dynamic memory disambiguation in Assignment 4) */
static struct reservation_station *LSQ; /* load/store queue */
static int LSQ_head, LSQ_tail; /* LSQ head and tail */
static int LSQ_num; /* num entries in LSQ */

Registers

/* Create Vector maps a logical register to a creator in the RUU (and specific output operand) or the architected register file (if RS_link is null) */
struct CV_link {
    struct reservation_station *rs; /* creator's RS */
    int odep_num; /* specific operand/register num */
};
static struct CV_link CVLINK_NULL = {NULL,0};
static struct CV_link create_vector[MD_TOTAL_REGS];

Referring to Reservation Stations

/* reservation station link: each RS_LINK node contains a pointer to the reservation_station entry it references along with an instance tag */
struct RS_link {
    struct RS_link *next;
    struct reservation_station *rs; /* referenced RS */
    INST_TAG_TYPE tag; /* instr tag */
    union {
        tick_t when; /* time stamp of entry (for eventq) */
        tick_t when; /* time when entry will be done (for eventq) */
        INST_SEQ_TYPE seq; /* instr seq no. (for readyq) */
    } x;
};

More Queues

/* Ready Instruction Queue contains instructions that have all of their *register* dependencies satisfied */
static struct RS_link *ready_queue;

/* pending event queue, sorted from soonest to latest event (in time): contains RS_link entries for instructions that are in execution */
static struct RS_link *event_queue;

Fetch (IF)

- Fetch instructions and put them in Fetch-Dispatch queue
Dispatch (ID1)

- Note that instruction are sent to issue in order.
- In each cycle, if the top instruction cannot be sent to the issue stage keep it as last instruction to work on.
- In each cycle see if there is a “last instruction” and deal with that; if not get a new one
- Perform functional simulation.
- For mispredicted branches squash all instructions in the IF/ID queue
  - \( \text{fetch}_\text{head} = \text{fetch}_\text{num} = 1; \text{fetch}_\text{tail} = 0; \)
  - Assuming that at the end of dispatch fetch_num is decremented and fetch_head is adjusted.
- Create RUU and LSQ entries if needed.
- Read operands:
  1. If no dependencies, then put it in ready queue.
  2. At the same time set the create_vector indicating you will produce the value of a register if needed.

Dispatch (Cont’d)

- Loads and stores are handled in a special way
  - Split the operation into two ops.
  - An add instruction for calculating the effective address installed in RUU.
  - An LD/ST operation installed in LSQ.
- Until the LD/ST effective address calculation is done and the LD/ST is put in ready queue, no other instruction will be issued.

Issue (ID2)

- Assign the top of ready queue to a FU.
- Schedule events such that we know when it will be done.

Issue (Cont’d)

- Issue of “store” is different; Marked as “completed” in the same cycle.
- No entry in events queue (eventq).
- No Write Back; “print” write in the same cycle if you wish.

Execute (EX)

- Using op latency of one:
  - IF ID IS WB CM.
- Using op latency of two:
  - IF ID IS EX WB CM.
- Using op latency of three:
  - IF ID IS EX EX WB CM.

Write Back (WB)

- release_fu();
- writeback();
- Check event queue if any instruction is done mark as complete and update the create vector.
- Note that FU is released even though the instruction may get stuck in WB (because WB is being used by another instruction).
  - In other words, as soon as an instruction spends “operation latency” number of cycles in ISSUE and EXEs, corresponding FU becomes available.
- Set the output registers as available (create_vector) here.
  - This means in the same cycle a register is written to, another instruction waiting for it can finish its Dispatch and get into ISSUE in the following cycle.

Note:

- An add instruction for calculating the effective address installed in RUU.
- An LD/ST operation installed in LSQ.
- Until the LD/ST effective address calculation is done and the LD/ST is put in ready queue, no other instruction will be issued.

- Issue of “store” is different; Marked as “completed” in the same cycle.
- No entry in events queue (eventq).
- No Write Back; “print” write in the same cycle if you wish.

- Does not consume WB resources; if previous instruction also in WB at the same cycle, there will be two in WB in same cycle. The output of pipe2 will only show one.
Commit

- Examine head of RRU queue until reaching an incomplete inst
- For these instructions
  - If store and there is port available do it and commit
  - If everything goes as planned release RUU and LSQ (if need be)
  - As soon as a problem (e.g. store cannot complete) stop for this cycle