G22.2243-001
High Performance Computer Architecture

Lecture 11
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

[Hennessy/Patterson *CA:AQA (4th Edition)*: Chapter 4]
Snooping - Cache State Machine: Combined

State machine for **CPU** requests for each **cache block** and for **bus** requests for each **cache block**

State Machine for CPU requests for each cache block and for bus requests for each cache block.

- **Invalid**
  - Write miss for this block
  - Write Back Block; (abort memory access)
  - CPU read hit
  - CPU write hit

- **Exclusive** (read/write)
  - CPU Write
  - Place Write Miss on bus
  - CPU read miss
  - Write back block, Place read miss on bus
  - CPU read hit
  - CPU write hit

- **Shared** (read only)
  - CPU Read
  - Place read miss on bus
  - CPU read miss
  - Place read miss on bus
  - CPU Read hit

- **Write Back** Block; (abort memory access)
  - Write Back Cache block
  - Place write miss on bus
Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- One Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - simpler protocol (centralized/one location)
  - directory is $f(\text{memory size} \times \text{number of processors})$ vs. $f(\text{cache size})$
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Processors have copies of their memory blocks and in what state
Distributed Directory MPs
Directory Protocol

• Similar to Snooping Protocol: Three states
  – **Shared**: ≥ 1 processors have data, memory up-to-date
  – **Uncached**: (no processor has it; not valid in any cache)
  – **Exclusive**: 1 processor (owner) has data; memory out-of-date

• In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)

• Keep it simple(r):
  – Writes to non-exclusive data
    → write miss
  – Processor blocks until access completes
  – Assume messages received and acted upon in order sent
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><strong>Read miss</strong></td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor P has a read miss at address A; request data and make P a read sharer</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write miss</strong></td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processor P has a write miss at address A; request data and make P exclusive owner</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Invalidate</strong></td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalidate a shared copy of data at address A</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fetch</strong></td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch block at address A &amp; send it to its home directory; change state to shared at remote</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fetch/Invalidate</strong></td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fetch block at address A &amp; send it to its home directory; invalidate the block in the cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data value reply</strong></td>
<td>Home directory</td>
<td>Local cache</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return a data value from the home memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data write-back</strong></td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write-back a data value for address A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
CPU - Cache State Machine

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

- Invalid (Uncached)
  - Fetch/Invalidate: send Data Write Back message to home directory
  - CPU Write: Send Write Miss msg to h.d.
  - CPU Read: Send Read Miss Message to h.d.
  - CPU Read hit: Invalidate

- Exclusive (read/writ)
  - CPU read hit
  - CPU write hit

- Shared (read only)
  - CPU read miss: Send Read Miss
  - CPU Write: Send Write Miss message to home directory
  - Fetch: send Data Write Back message to home directory
  - Fetch/Invalidate: send Data Write Back message to home directory

- CPU write miss: send Data Write Back message and Write Miss to home directory

CPU read miss: send Data Write Back message and read miss to home directory
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.
• 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

**Uncached**
- Data Write Back: Sharers = {}
  
  *(Write back block)*

**Exclusive**
- Write Miss: Sharers = {P}; send Data Value Reply msg

**Shared**
- Write Miss: Sharers = {P}; send Data Value Reply msg
- Write Miss: send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
- Read miss: Sharers += {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache *(Write back block)*

**Read miss:** Sharers += {P}; send Data Value Reply

*(Write back block)*
# Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Bus</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
### Example

#### Processor 1  Processor 2  Interconnect  Directory  Memory

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
<td>P2</td>
<td>Bus</td>
<td></td>
<td>Memor</td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>Addr</td>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>State</td>
<td>Addr</td>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>A1</td>
<td></td>
<td>Procs</td>
<td>{P1}</td>
</tr>
<tr>
<td></td>
<td>A1</td>
<td>10</td>
<td></td>
<td>Ex</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DaRp</td>
<td>P1</td>
<td>A1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<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><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td><strong>WrMs</strong></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td><strong>DaRp</strong></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
Example

### Processor 1 Processor 2 Interconnect Directory Memory

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Bus</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>A1</td>
<td></td>
<td>Ex</td>
<td>{P1}</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
<td>RdMs</td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>Ftch</td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>DaRp</td>
<td>P2</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td>P1</td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td>P2</td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>{P1,P2}</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Write Back**

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><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
<td>WrMs</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv.</td>
<td></td>
<td></td>
<td>Inval.</td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block
Example

A1 and A2 map to the same cache block

<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><strong>step</strong></td>
<td><strong>State</strong></td>
<td><strong>Addr</strong></td>
<td><strong>Value</strong></td>
<td><strong>Action</strong></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
</tr>
<tr>
<td>P2: Read 20 to A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>Ftch</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv.</td>
<td>P1</td>
<td>A1</td>
<td>Inval.</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A2</td>
<td>20</td>
<td>WrBk</td>
</tr>
<tr>
<td>A1 and A2 map to the same cache block</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
• Optimization:
  – 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 MPs, 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 register to 1 & swap
  
  − New value in register 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
• 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:

```assembly
try:  mov   R3,R4       ; mov exchange value
   ll    R2,0(R1)    ; load linked
   sc    R3,0(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:

```assembly
try:  ll    R2,0(R1)  ; load linked
   addi  R2,R2,#1    ; increment (OK if reg–reg)
   sc    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

  ```asm
  li    R2,#1
  lockit:     exch  R2,0(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”):

  ```asm
  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

\[ \begin{align*}
\text{P1: } A &= 0; \\
\text{...} &
\text{A} = 1; \\
\text{L1: } &\text{if (B == 0) ...} \\
\text{P2: } B &= 0; \\
\text{...} &
\text{B} = 1; \\
\text{L2: } &\text{if (A == 0) ...}
\end{align*} \]

- 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
    - write (x)
    - ... release (s) \{unlock\}
    - ... acquire (s) \{lock\}
    - ... read(x)
- 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/d^2)
  – 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 nxn grid: O(n^2)
  – Input: 130 x 130 grid points, 5 iterations
FFT and LU Kernels

- 512 KB two-way data cache with 64-byte blocks
Barnes and Ocean Apps

Barnes

Ocean

Average cycles per reference

Processor count

8 16 32 64

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5

Remote miss

Local miss

Cache hit

3-hop miss to remote cache
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(n^3), 100 P?
  - **Time-constrained scaling**: 1 hour, => 10^{1/3}n => 2.15n scale up
  - **Memory-constrained scaling**: 10n size => 10^3/10 => 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

• Multilevel cache hierarchy + multilevel 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
Pitfall: 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
- 1 GB, uni = $138k v. mp = $181k + $20k * P
- What speedup for better MP cost performance?
  - 8 proc $\rightarrow$ $341k$; $341k/138k$ => 2.5X
  - 16 proc $\rightarrow$ 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), DSS 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 (2\textsuperscript{nd} cycle)
    • EXE (3\textsuperscript{rd} 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 oplat;      /* 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
  - `fetch_head = (FETCH_QUEUE_SIZE - 1); fetch_num = 1; fetch_tail = 0;`
  - Assuming that at the end of dispatch `fetch_num` is decremented and `fetch_head` is adjusted
- Create RUU and LSQ entries if need be
- 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 need be
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

```c
fu = res_get( fu_pool, MD_OP_FUCLASS(rs->op) );
if ( fu )
{
    /* can issue instruction */
    readyq_pop();
    rs->issued = TRUE;
    /* schedule when next instruction can be issued to this FU */
    fu->master->busy = fu->issuelat;  // WILL BE RESET IN WB ( release_fu() )
    /* schedule a result writeback event */
    eventq_queue_event( rs, sim_num_cycles + fu->oplat );
}
```
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
  ```c
  if ( rs->in_LSQ &&
      ((MD_OP_FLAGS(rs->op) & (F_MEM|F_STORE)) == (F_MEM|F_STORE)) ) {
      /* instruction is a store */
      .....  
      if ( verbose ) print_verbose_message( "write", rs );
  }
  ```
- 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.
Execute (EX)

• Using op latency of one:
  – IF ID IS WB CM
    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
    
    | IF | ID | IS | WB | CM | producer |
    |----|----|----|----|----|---------|
    |    |    | IF | ID | IS | WB | CM | consumer |

11/7/2007
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