Cache Review

- Three basic models and how they work
  - Direct mapped, fully associative, set associative

- Cache Coherence
## Memory Hierarchy: Apple iMac G5

### Managed by compiler

<table>
<thead>
<tr>
<th>07</th>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
<td></td>
</tr>
</tbody>
</table>

### Managed by hardware

<table>
<thead>
<tr>
<th>07</th>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
<td></td>
</tr>
</tbody>
</table>

### Managed by OS, hardware, application

<table>
<thead>
<tr>
<th>07</th>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
<td></td>
</tr>
</tbody>
</table>

### iMac G5

- **1.6 GHz**
- **Latency (Cycles, Time):**
  - 0.6 ns, 1.9 ns, 1.9 ns, 6.9 ns, 55 ns, 12 ms
- **Size:** 1K, 64K, 32K, 512K, 256M, 80G

---

**Managed by compiler:**
- Managed by compiler

**Managed by hardware:**
- Managed by hardware

**Managed by OS, hardware, application:**
- Managed by OS, hardware, application
iMac’s PowerPC 970: All caches on-chip

- L1 (64K Instruction)
- L1 (32K Data)
- Registers
- 512K L2

(1K)
Memory Hierarchy: Terminology

° **Hit**: data appears in some block in the upper level
  - **Hit Rate**: the fraction of memory access found in the upper level
  - **Hit Time**: Time to access the upper level which consists of
    Time to determine hit/miss + time to deliver block to processor

° **Miss**: data needs to be retrieve from a block in the lower level
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Time**: Time to determine hit/miss + Time to replace a block in the upper level + Time to deliver the block to the processor
  - **Miss Penalty**: Extra time incurred for a miss = Time to replace a block in the upper level

° **Hit Time << Miss Penalty and Miss Time**
Block 12 placed in 8 block cache:
• Fully associative, direct mapped, 2-way set associative
Example: 1 KB Direct Mapped Cache with 32 B Blocks

° For a $2^N$ byte cache:
  - The uppermost ($32 - N$) bits are always the Cache Tag
  - The lowest $M$ bits are the Byte Select (Block Size = $2^M$)

```

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Example: 0x50</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ex: 0x01</td>
<td>Ex: 0x00</td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit Cache Tag Cache Data

0x50

<table>
<thead>
<tr>
<th>Byte 31</th>
<th>**</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 63</td>
<td>**</td>
<td>Byte 33</td>
<td>Byte 32</td>
</tr>
<tr>
<td>Byte 1023</td>
<td>**</td>
<td>Byte 992</td>
<td></td>
</tr>
</tbody>
</table>

```

Stored as part of the cache “state”
Example: Fully Associative

° Fully Associative Cache
  • Forget about the Cache Index
  • Compare the Cache Tags of all cache entries in parallel
  • Example: Block Size = 32 B blocks, we need N 27-bit comparators
**Example: Set Associative Cache**

*N-way set associative*: N entries for each Cache Index
- N blocks per set; N caches operate in parallel

**Example: Two-way set associative cache**
- Cache Index selects a “set” from the cache (16 sets if 32 blocks, 4 bits)
- The two tags in the set are compared to the input in parallel
- Data is selected based on the tag result
Q2: How is a block found if it is in the upper level?

- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag
Cache Line Replacement Algorithms

° Deterministic for Direct Mapped (only one choice)

° Set Associative or Fully Associative:
  • Random
  • LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Writing in Caches

° **Writes occur less frequently than reads:**
  • Under MIPS: 7% of all memory traffic are writes
  • 25% of all data traffic are writes

° Thus, Amdahl’s Law implies that caches should be optimized for reads. However, we cannot ignore writes.

° **Problems with writes:**
  • Must check tag BEFORE writing into the cache
  • Only a portion of the cache block is modified
  • Write stalls - CPU must wait until the write completes
Design Options for Write

° **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.

° **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  
  • is block clean or dirty?

° Pros and Cons of each?
  
  • WT: read misses don’t cause writes; easier to implement; copy of data always exists
  
  • WB: write at the speed of the cache; multiple writes to cache before write to memory; less memory BW consumed
A Write Buffer is needed between the Cache and Memory

- Processor: writes data into the cache and the write buffer
- Memory controller: write contents of the buffer to memory

Write buffer is just a FIFO:
- Typical number of entries: 4
- Works fine if: Store frequency (w.r.t. time) $<< 1 /$ DRAM write cycle

Memory system designer’s nightmare:
- Store frequency (w.r.t. time) $> 1 /$ DRAM write cycle
- Write buffer saturation
Write Buffer Saturation

- **Store frequency (w.r.t. time) > 1 / DRAM write cycle**
  - If this condition exist for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - The CPU Cycle Time <= DRAM Write Cycle Time

- **Solution for write buffer saturation:**
  - Use a write back cache
  - Install a second level (L2) cache: (does this always work?)
Write Miss Design Options

° Write allocate ("fetch on write") - block is loaded on a write miss, followed by the write.

° No-write allocate ("write around") - block is modified in the lower level, not loaded into the cache.
2 Classes of Cache Coherence Protocols

1. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   • All caches are accessible via some broadcast medium (a bus or switch)
   • All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

2. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
CACHE COHERENCE Requirements

° Provide set of states, state transition diagram, and actions

° Manage coherence protocol
  • (0) Determine when to invoke coherence protocol
  • (a) Find info about state of block in other caches to determine action
    - whether need to communicate with other cached copies
  • (b) Locate the other copies
  • (c) Communicate with those copies (invalidate/update)

° (0) is done the same way on all systems
  • state of the line is maintained in the cache
  • protocol is invoked if an access fault occurs on the line

° Different approaches distinguished by (a) to (c)
Bus-based Coherence

° All of (a), (b), (c) done through broadcast on bus
  • faulting processor sends out a “search”
  • others respond to the search probe and take necessary action

° Could do it in scalable network too
  • broadcast to all processors, and let them respond

° Conceptually simple, but broadcast doesn’t scale with p
  • on bus, bus bandwidth doesn’t scale
  • on scalable network, every fault leads to at least p network transactions

° Scalable coherence:
  • can have same cache states and state transition diagram
  • different mechanisms to manage protocol
Snoopy Cache-Coherence Protocols

° Cache Controller “snoops” all transactions on the shared medium (bus or switch)
  • relevant transaction if for a block it contains
  • take action to ensure coherence
  • depends on state of the block and the protocol

° Either get exclusive access before write via write invalidate or update all copies on write
Scalable Approach: Directories

• Every memory block has associated directory information
  – keeps track of copies of cached blocks and their states
  – on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  – in scalable networks, communication with directory and copies is through network transactions

• Many alternatives for organizing directory information
Basic Operation of Directory

- **k** processors.
- With each cache-block in memory: 
  - **k** presence-bits, 1 dirty-bit
- With each cache-block in cache: 
  - 1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i; }

- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - ...
  - ...

- 5/12/11
CPU - Cache State Machine

Invalid

- CPU Read hit
- Send Read Miss message
- CPU Write: Send Write Miss msg to home directory
- <Fetch/Invalidate>: send Data Write Back message to home directory

Exclusive (read/write)

- CPU Read hit
- CPU write hit

Shared (read/only)

- CPU read miss: Send Read Miss message to home directory
- CPU write miss: Send Write Miss message to home directory
- <Fetch>: send Data Write Back message to home directory
- CPU read miss: send Data Write Back message and read miss to home directory
- CPU write miss: send Data Write Back message and Write Miss to home directory