CS352H: Computer Systems Architecture

Topic 11: Memory Hierarchy - Caches

October 13, 2009
Memory Technology

- **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

- **Dynamic RAM (DRAM)**
  - 50ns – 70ns, $20 – $75 per GB

- **Magnetic disk**
  - 5ms – 20ms, $0.20 – $2 per GB

- **Ideal memory**
  - Access time of SRAM
  - Capacity and cost/GB of disk
Principle of Locality

- Programs access a small proportion of their address space at any time

- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables

- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data
Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
  - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU
Memory Hierarchy Levels

- **Block (aka line):** unit of copying
  - May be multiple words
- **If accessed data is present in upper level**
  - **Hit:** access satisfied by upper level
    - Hit ratio: hits/accesses
- **If accessed data is absent**
  - **Miss:** block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      \[ = 1 - \text{hit ratio} \]
  - Then accessed data supplied from upper level
Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
  - Given accesses $X_1, \ldots, X_{n-1}, X_n$

How do we know if the data is present? Where do we look?

<table>
<thead>
<tr>
<th>$X_4$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_{n-2}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_{n-1}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_3$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$X_4$</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_{n-2}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_{n-1}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_3$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Before the reference to $X_n$  

b. After the reference to $X_n$
Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - (Block address) modulo (#Blocks in cache)

#Blocks is a power of 2
Use low-order address bits
Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0
Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Cache Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10 110</td>
<td>Miss</td>
<td>110</td>
</tr>
</tbody>
</table>

## Index

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>110</strong></td>
<td>Y</td>
<td><strong>10</strong></td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Cache Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>11 010</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Cache Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10 110</td>
<td>Hit</td>
<td>110</td>
</tr>
<tr>
<td>26</td>
<td>11 010</td>
<td>Hit</td>
<td>010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Cache Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10 000</td>
<td>Miss</td>
<td>000</td>
</tr>
<tr>
<td>3</td>
<td>00 011</td>
<td>Miss</td>
<td>011</td>
</tr>
<tr>
<td>16</td>
<td>10 000</td>
<td>Hit</td>
<td>000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10</td>
<td>Mem[10000]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Mem[00011]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Cache Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>10 010</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10</td>
<td>Mem[10000]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>010</strong></td>
<td>Y</td>
<td>10</td>
<td><strong>Mem[10010]</strong></td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Mem[00011]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Address Subdivision

Address (showing bit positions)

31 30 ⋯ 13 12 11⋯ 2 1 0

Byte offset

Hit

Tag

Index

Data

Valid

Index

0
1
2
⋯

⋯

⋯

1021
1022
1023

20

32

=
Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
  - Block address = \[1200/16\] = 75
  - Block number = 75 modulo 64 = 11

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 bits</td>
<td>6 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality
- But in a fixed-sized cache
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
    - Larger blocks ⇒ pollution
- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help
Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full
Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
Write Allocation

- What should happen on a write miss?

- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)

- For write-back
  - Usually fetch the block
Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks \( \times \) 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%
Example: Intrinsity FastMATH
Main Memory Supporting Caches

- Use DRAMs for main memory
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- Example cache block read
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- For 4-word block, 1-word-wide DRAM
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle
Increasing Memory Bandwidth

4-word wide memory
- Miss penalty = 1 + 15 + 1 = 17 bus cycles
- Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle

4-bank interleaved memory
- Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
- Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle
Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs
DRAM Generations

<table>
<thead>
<tr>
<th>Year</th>
<th>Capacity</th>
<th>$/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64Kbit</td>
<td>$1500000</td>
</tr>
<tr>
<td>1983</td>
<td>256Kbit</td>
<td>$500000</td>
</tr>
<tr>
<td>1985</td>
<td>1Mbit</td>
<td>$200000</td>
</tr>
<tr>
<td>1989</td>
<td>4Mbit</td>
<td>$50000</td>
</tr>
<tr>
<td>1992</td>
<td>16Mbit</td>
<td>$15000</td>
</tr>
<tr>
<td>1996</td>
<td>64Mbit</td>
<td>$10000</td>
</tr>
<tr>
<td>1998</td>
<td>128Mbit</td>
<td>$4000</td>
</tr>
<tr>
<td>2000</td>
<td>256Mbit</td>
<td>$1000</td>
</tr>
<tr>
<td>2004</td>
<td>512Mbit</td>
<td>$250</td>
</tr>
<tr>
<td>2007</td>
<td>1Gbit</td>
<td>$50</td>
</tr>
</tbody>
</table>
Measuring Cache Performance

- **Components of CPU time**
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- **With simplifying assumptions:**

  Memory stall cycles

  \[
  \text{Memory accesses} = \frac{\text{Program}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}
  \]

  \[
  = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}
  \]
Cache Performance Example

- **Given**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions

- **Miss cycles per instruction**
  - I-cache: $0.02 \times 100 = 2$
  - D-cache: $0.36 \times 0.04 \times 100 = 1.44$

- **Actual CPI** = $2 + 2 + 1.44 = 5.44$
  - Ideal CPU is $5.44/2 = 2.72$ times faster
Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction
Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance
Associative Caches

- **Fully associative**
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)

- **n-way set associative**
  - Each set contains $n$ entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - $n$ comparators (less expensive)
Associative Cache Example

**Direct mapped**

- Block #: 0, 1, 2, 3, 4, 5, 6, 7
- Data
- Tag: 1, 2
- Search

**Set associative**

- Set #: 0, 1, 2, 3
- Data
- Tag: 1, 2
- Search

**Fully associative**

- Data
- Tag: 1, 2
- Search
Spectrum of Associativity

For a cache with 8 entries

One-way set associative (direct mapped)

Two-way set associative

Four-way set associative

Eight-way set associative (fully associative)
Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>miss</td>
<td>Mem[0] Mem[6]</td>
</tr>
</tbody>
</table>
# Associativity Example

## 2-way set associative

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>hit</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
</tbody>
</table>

## Fully associative

<table>
<thead>
<tr>
<th>Block address</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Set 0</td>
</tr>
<tr>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>0</td>
<td>hit</td>
<td>Mem[0]</td>
</tr>
</tbody>
</table>
How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
Set Associative Cache Organization
Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity
3C Model of Cache Behavior

- **Compulsory misses**: These are caused by the first access to a new block. They are also called **cold-start misses**.
- **Conflict misses**: These occur in non-full-associative caches when multiple blocks compete for the same set. These are also called **collision misses**. They are the misses that would be eliminated by use of a fully associative cache.
- **Capacity misses**: These are caused when the cache is too small to contain all the blocks needed during execution. They are non-conflict misses that occur when blocks are replaced and later retrieved.

![Graph showing miss rate per type vs cache size](image)

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Possible negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decreases capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decreases miss rate due to conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decreases miss rate for a wide range of block sizes due to spatial locality</td>
<td>Increases miss penalty. Very large block could increase miss rate</td>
</tr>
</tbody>
</table>
Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

<table>
<thead>
<tr>
<th>31</th>
<th>10</th>
<th>9</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 bits</td>
<td>10 bits</td>
<td>4 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interface Signals

- CPU to Cache:
  - Read/Write
  - Valid
  - Address (32 bits)
  - Write Data (32 bits)
  - Read Data (32 bits)
  - Ready

- Cache to Memory:
  - Read/Write
  - Valid
  - Address (32 bits)
  - Write Data (128 bits)
  - Read Data (128 bits)
  - Ready

- Multiple cycles per access
Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state $= f_n$ (current state, current inputs)
- Control output signals $= f_o$ (current state)
Cache Controller FSM

Could partition into separate states to reduce clock cycle time
Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache
Multilevel Cache Example

- **Given**
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns

- **With just primary cache**
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 \times 400 = 9
Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 500 cycles
- CPI = $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = $9/3.4 = 2.6$
Multilevel Cache Considerations

- **Primary cache**
  - Focus on minimal hit time

- **L-2 cache**
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact

- **Results**
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size
Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation
Interactions with Software

- Misses depend on memory access patterns
- Algorithm behavior
- Compiler optimization for memory access
Cache Coherence Problem

- Suppose two CPU cores share a physical address space
- Write-through caches

<table>
<thead>
<tr>
<th>Time step</th>
<th>Event</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)  
    ⇒ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)  
    ⇒ read returns written value
  - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X  
    ⇒ all processors see writes in the same order
  - End up with the same final value for X
Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access

- Snooping protocols
  - Each cache monitors bus reads/writes

- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory
Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

<table>
<thead>
<tr>
<th>CPU activity</th>
<th>Bus activity</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU A writes 1 to X</td>
<td>Invalidate for X</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B read X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Consistency

- When are writes seen by other processors
  - “Seen” means a read returns the written value
  - Can’t be instantaneously

- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses

- Consequence
  - P writes X then writes Y
    \[ \Rightarrow \text{all processors that see new Y also see new X} \]
  - Processors can reorder reads, but not writes
Multilevel On-Chip Caches

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 256KB L2 cache
# 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong> (per core)</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong> (per core)</td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>L3 unified cache</strong> (shared)</td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
</tbody>
</table>

n/a: data not available
Miss Penalty Reduction

- Return requested word first
  - Then back-fill rest of block

- Non-blocking miss processing
  - Hit under miss: allow hits to proceed
  - Miss under miss: allow multiple outstanding misses

- Hardware prefetch: instructions and data

- Opteron X4: bank interleaved L1 D-cache
  - Two concurrent accesses per cycle
Pitfalls

- **Byte vs. word addressing**
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4

- **Ignoring memory system effects when writing or generating code**
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality
Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores ⇒ need to increase associativity
- Using AMAT to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation
Concluding Remarks

- Fast memories are small, large memories are slow
  - We really want fast, large memories 😞
  - Caching gives this illusion 😊
- Principle of locality
  - Programs use a small part of their memory space frequently
- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- Memory system design is critical for multiprocessors