Memory Hierarchy

Reading

- Sections 5.1, 5.2, 5.3, 5.4, 5.8 (some elements), 5.9
Memories: Two Basic Types

- **SRAM:**
  - Value is stored on a pair of inverting gates
  - Very fast but takes up more space than DRAM (4 to 6 transistors)

- **DRAM:**
  - Value is stored as a charge on capacitor (must be refreshed)
  - Very small but slower than SRAM (factor of 5 to 10)
Memory Technology

- Registers
  - Integrated with the CPU: fastest and most expensive

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

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

- Magnetic disk
  - 5ms – 20ms, $0.05 – $0.50 per GB

- Ideal memory
  - Access time of register
  - Capacity and cost/GB of disk

These numbers keep changing fast!

The Memory Hierarchy

Where do Solid State Disks (SSDs) fit?
Memory Hierarchy

- Going off-chip is expensive in time and energy

The Memory Wall

"Multicore Is Bad News For Supercomputers"
IEEE Spectrum 2008

- Data intensive applications
- Memory bandwidth demand is scaling faster than memory interface capacity

"You can buy bandwidth but you cannot bribe God"
- unknown
Key Driver is Energy/Power

Embedded Platforms

Goal: 1-100 GOps

Big Science: To Exascale

Goal: 20MW/Exaflop

Cost of Data Movement

• Data movement becomes more expensive (energy) than computation!

(9)

Data movement becomes more expensive than computation!

(10)

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 and virtual memory concept
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
  - Cache memory attached to CPU
- Copy most recently accessed items from cache to registers

Not shown - the stack!
Cache Basic Concepts

- **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
      \[ = 1 - \text{hit ratio} \]

Cache Memory

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

| \( x_4 \) | \( X_4 \) |
| \( x_1 \) | \( X_1 \) |
| \( x_{n-2} \) | \( X_{n-2} \) |
| \( x_{n-1} \) | \( X_{n-1} \) |
| \( x_0 \) | \( X_0 \) |
| \( x_5 \) | \( X_5 \) |

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

a. Before the reference to \( X_p \)  
b. After the reference to \( X_p \)
Basic Principle: Address Breakdown

- Page #/Page address
- Byte within a page

32-bit word

- 0x80080000
- 0x80080004
- 16 byte line
- 4KB page
- 0x80081000

- Line #/address
- Word in a line
- Byte in a line

- 28
- 2
- 2

Same address can be interpreted in more than one way

Examples:

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>

<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>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>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>

(21)

### 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>

(22)
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>010</td>
<td>Y</td>
<td>10</td>
<td>Mem[10010]</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

Index

Tag

20

10

Hit

Data

Valid

Tag

Index

0

1

2

···

···

···

1021

1022

1023

20

32
Block Size Considerations

- Larger blocks should reduce miss rate
  - Due to spatial locality

- But in a fixed-sized cache
  - Larger blocks \(\Rightarrow\) fewer of them
    - More competition \(\Rightarrow\) increased miss rate
  - Larger blocks \(\Rightarrow\) pollution

- Larger miss penalty
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

---

Performance

- Increasing the block size tends to decrease miss rate:

---

<table>
<thead>
<tr>
<th>Program</th>
<th>Block size in words</th>
<th>Instruction miss rate</th>
<th>Data miss rate</th>
<th>Effective combined miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1</td>
<td>6.1%</td>
<td>2.1%</td>
<td>5.4%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.0%</td>
<td>1.7%</td>
<td>1.9%</td>
</tr>
<tr>
<td>spice</td>
<td>1</td>
<td>1.2%</td>
<td>1.3%</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>
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 Through (cont.)

- Write buffers are used to hide the latency of memory writes by overlapping writes with useful work.
- Ensures consistency between cache contents and main memory contents at all times.
- Write traffic can dominate performance.

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.
- Still use the write buffer to hide the latency of write operations.
Write Back (cont.)

- Locality of writes impacts memory traffic
- Writes occur at the speed of a cache
- Complexity of cache management is increased
- Cache may be inconsistent with main memory

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
Summary: Hits vs. Misses

- **Read hits**
  - This is what we want!

- **Read misses**
  - Stall the CPU, fetch block from memory, deliver to cache, restart

- **Write hits:**
  - Can replace data in cache and memory (write-through)
  - Write the data only into the cache (write-back the cache later)

- **Write misses:**
  - Read the entire block into the cache, then write the word... ?

---

Interface Signals

- **CPU**
  - Read/Write
  - Valid
  - Address \( \text{32} \)
  - Write Data \( \text{32} \)
  - Read Data \( \text{32} \)
  - Ready

- **Cache**
  - Read/Write
  - Valid
  - Address \( \text{32} \)
  - Write Data \( \text{128} \)
  - Read Data \( \text{128} \)
  - Ready

- **Memory**
  - Read/Write
  - Valid
  - Address \( \text{32} \)
  - Write Data
  - Read Data

Multiple cycles per access
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%
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
  - Send address(es) to memory
  - Time to read a cache line
  - Time to transfer data to the cache
• Consider all of the steps a `lw` instruction must go through!
• We will use a simple model

System Organization

- SATA Connector (x4)
- BIOS Flash Chip in PLCC Socket
- Southbridge (with heatsink)
- CMOS Backup Battery
- Integrated graphics processor (with heatsink)
- PCI Slot (x3)
- Integrated audio codec chip
- Integrated Gigabit Ethernet chip
- Connectors for Integrated Peripherals
  - PCI Express Slot
  - Floppy Drive Connector
  - iDE Connector (x2)
  - Super IO Chip
  - 24-pin ATX Power Connector
  - DIMM Memory Slots (x4)
  - CPU Fan Connector
  - CPU Fan & Heatsink Mount
  - CPU Socket (Socket 939)
Basic DRAM Organization

Example: Micron 1Gb SDRAM

From https://www.sei.cmu.edu/cyber-physical/research/timing-verification/Multicore-scheduling-cont.cfm
Increasing Memory Bandwidth

- Example cache block read for organization a.
  - 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 = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle
Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- Computer memory stall cycles

---

Measuring Performance

Memory Stall Cycles

- Read Stalls: 
  - IC * Reads/Instruction * read miss rate * miss penalty

- Write Stalls: 
  - IC * writes/Instruction * write miss rate * miss penalty

- IC * memory references/Instruction * miss rate * miss penalty

- Instructions * references/Instruction
- Data references + Instruction references

- These expressions themselves are an approximation
- Note the equivalence between the use of misses/instruction and misses/memory reference
- Some Example Problems
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 $\times$ Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = $1 + 0.05 \times 20 = 2$ns
    - 2 cycles per instruction

Increase in CPI = Base CPI + Prob(event) * Penalty(event)

- Examples
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
    - \((\text{Block number}) \mod \text{(#Sets in cache)})
  - Search all entries in a given set at once
  - \( n \) comparators (less expensive)
Example: Fully Associative Cache

Spectrum of Associativity

- For a cache with 8 entries

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Eight-way set associative (fully associative)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>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[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[0]</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>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[0]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] 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>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>miss</td>
<td>Mem[0] Mem[0]</td>
</tr>
<tr>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>miss</td>
<td>Mem[0] Mem[0] Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>hit</td>
<td>Mem[0] Mem[0] 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
Summary: Placement Policy

- Direct Mapped
  - No choice

- Set Associative
  - Any location in the set of lines
  - Replacement policy

- Fully Associative
  - Any line in the cache
  - Dictated by the replacement policy

Summary: 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
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 Caches (cont.)

- Goal: balance (fast) hits vs. (slow) misses
  - Techniques for the former are distinct from those for the latter

- Goal: keep up with the processor vs. keep up with memory

Example: Addressing
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

Sources of Misses

• Compulsory misses (aka cold start misses)
  ❖ First access to a block

• Capacity misses
  ❖ Due to finite cache size
  ❖ A replaced block is later accessed again

• Conflict misses (aka collision misses)
  ❖ In a non-fully associative cache
  ❖ Due to competition for entries in a set
  ❖ Would not occur in a fully associative cache of the same total size
### Cache Design Trade-offs

<table>
<thead>
<tr>
<th>Design change</th>
<th>Effect on miss rate</th>
<th>Negative performance effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase cache size</td>
<td>Decrease capacity misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase associativity</td>
<td>Decrease conflict misses</td>
<td>May increase access time</td>
</tr>
<tr>
<td>Increase block size</td>
<td>Decrease compulsory misses</td>
<td>Increases miss penalty. For very large block size, may increase miss rate due to pollution.</td>
</tr>
</tbody>
</table>

### Miss Penalty Reduction

- Return requested word first
  - Then back-fill rest of block
- Non-blocking miss processing
  - Hit under miss: allow hits to proceed
  - Mis under miss: allow multiple outstanding misses
- Hardware prefetch: instructions and data
- Opteron X4: bank interleaved L1 D-cache
  - Two concurrent accesses per cycle
Example: Intel Sandy Bridge

- Sandy Bridge i5-2400
  - L1 I & D cache: 32K, 8-way, 64 byte line
  - L2 unified cache: 256K, 8 way, 64 byte line
  - L3 Shared: 6MB, 12-way 64 byte line

- Sandy Bridge i7-970
- Sandy Bridge-E can have up to 20MB L3!

Example: Intel Nehalem

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB 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>L1 caches</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>(per core)</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>L2 unified cache</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>(per core)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 unified cache</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>
<tr>
<td>(shared)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n/a: data not available</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimizations for memory access
  - Think matrices vs. hash tables
- Analysis of memory access behavior is critical
- What about instruction scheduling?
Cache Coherence

- A shared variable may exist in multiple caches
- Multiple copies to improve latency
- This is a really a synchronization problem

![Diagram showing cache coherence](image)

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>0</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>

(71)

(72)
Example: Communicating Threads

Producer
while (1) {
  while (count == 0)
    ; // do nothing
  // add item to the buffer
  ++count;
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
}

Consumer
while (1) {
  while (count == BUFFER_SIZE)
    ; // do nothing
  // remove item from the buffer
  --count;
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}

Example (Writeback Cache)

Courtesy H. H. Lee
Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - \( P \) writes \( X \); \( P \) reads \( X \) (no intervening writes) \( \Rightarrow \) read returns written value
  - \( P_1 \) writes \( X \); \( P_2 \) reads \( X \) (sufficiently later) \( \Rightarrow \) read returns written value
    - c.f. CPU B reading \( X \) after step 3 in example
  - \( P_1 \) writes \( X \), \( P_2 \) writes \( X \) \( \Rightarrow \) 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></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<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>1</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>

(77)

Scalable Coherence

- What about large systems that cannot be connected with buses?
  - Imagine 256 cores on a chip or rack scale systems
- A shared address space exists across all cores
- Message-based communication protocols are used to enforce coherence
  - ECE 4100/6100 if you want to know more!

(78)
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
- Memory system design is critical for multiprocessors

Study Guide

- Given a memory system description, e.g., cache and DRAM parameters, what is the breakdown of the addresses?
- Given the state of the memory hierarchy be able to determine the changes required on a new access. See sample problems.
- Given a main memory and cache architecture, be able to compute the impact on CPI. See sample problems
- Given the state of a cache system in a coherent shared memory architecture be able to determine the state changes when a new access is provided
### Glossary

- Associativity
- Cache coherence
- Cache line or block
- Cache hit
- Cache miss
- Direct mapped cache
- Fully associative cache
- Memory hierarchy
- Multilevel cache
- Miss penalty
- Replacement policy
- Set associative cache
- Spatial locality
- Snooping protocol
- Temporal locality
- Tag
- Write through
- Write back