Chapter 5
Large and Fast: Exploiting Memory Hierarchy
Review: Major Components of a Computer

- Processor
  - Control
  - Datapath
- Memory
- Devices
  - Input
  - Output
- Cache
- Main Memory
- Secondary Memory (Disk)
Processor-Memory Performance Gap

- **"Moore’s Law"**
  - Processor (µProc): 55%/year (2X/1.5yr)
  - DRAM: 7%/year (2X/10yrs)

- Processor-Memory Performance Gap (grows 50%/year)
The "Memory Wall"

- Processor vs DRAM speed disparity continues to grow

- Good memory hierarchy (cache) design is increasingly important to overall performance
The Memory Hierarchy Goal

- Fact: Large memories are slow and fast memories are small

- How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?
  - With hierarchy
  - With parallelism
A Typical Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology.

```
Speed (%cycles): ½’s  1’s  10’s  100’s  10,000’s
Size (bytes):  100’s  10K’s  M’s  G’s  T’s
Cost:     highest  lowest
```
Memory Technology

- **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $500 – $1000 per GB (2012)

- **Dynamic RAM (DRAM)**
  - 50ns – 70ns, $10 – $20 per GB

- **Flash semiconductor**
  - 5,000-50,000 ns, $0.75 - 1.00 per GB

- **Magnetic disk**
  - 5ms – 20ms, $0.05 – $0.10 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
Characteristics of the Memory Hierarchy

- Increasing distance from the processor in access time

- Processor
  - L1:
    - 4-8 bytes (word)
  - L2:
    - 8-32 bytes (block)
  - Main Memory:
    - 1 to 4 blocks
  - Secondary Memory:
    - 1,024+ bytes (disk sector = page)

- Inclusive—what is in L1 is a subset of what is in L2 is a subset of what is in MM that is a subset of what is in SM

- (Relative) size of the memory at each level
DRAM Technology

- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM “row”
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>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>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>
<tr>
<td>2014</td>
<td>4 Gbit</td>
<td>$0.50</td>
</tr>
</tbody>
</table>
DRAM Performance Factors

- Row buffer
  - Allows several words to be read and refreshed in parallel

- Synchronous DRAM
  - Allows for consecutive accesses in bursts without needing to send each address
  - Improves bandwidth

- DRAM banking
  - Allows simultaneous access to multiple DRAMs
  - Improves bandwidth
Increasing Memory Bandwidth

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

- **4-bank interleaved memory (c)**
  - Miss penalty = 1 + 15 + 4×1 = 20 bus cycles
  - Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle
Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)
Flash Types

- **NOR flash**: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems

- **NAND flash**: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, ...

- Flash bits wears out after 30K-1000Ks of program/erase cycles
  - Wear leveling: remap data to less used blocks
Disk Storage

- Nonvolatile, rotating magnetic storage
Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps

- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead
Disk Access Example

- **Given**
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk

- **Average read time**
  - 4ms seek time
    + $\frac{1}{2} / (15,000/60) = 2\text{ms}$ rotational latency
    + $512 / 100\text{MB/s} = 0.005\text{ms}$ transfer time
    + $0.2\text{ms}$ controller delay
  - $= 6.2\text{ms}$

- If actual average seek time is 1ms
  - Average read time = $3.2\text{ms}$
Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times

- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
    - SCSI, ATA, SATA

- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay
Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 23
Cache Size

Increasing cache size

hit rate

1/(cycle time)

optimum

Increasing cache size
# Cache Memory

- **Cache memory**
  - The level of the memory hierarchy closest to the CPU

- Given accesses $X_1, \ldots, X_{n-1}, X_n$

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

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

---

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

\[10001_2 \text{ modulo } 1000_2 = 001_2\]
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 (sequential slides)

- 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

(sequential slides)

<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><strong>110</strong></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

(sequential slides)

<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

(sequential slides)

<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

(sequential slides)

<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 (sequential slides)

<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><strong>10</strong></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>

*replace*
Address Subdivision

Address (showing bit positions)

31 30  ⋮  13 12 11  ⋮  2  1  0

Byte offset

Hit

Tag

Index

Index

Valid

Tag

Data

... 0

1

2

... 1021

1022

1023

20

10

20

32

Data

4 bytes at a time
Example: Larger Block Size

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

<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 $\Rightarrow$ 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
Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)
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 \times 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 assume that
  - 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
- With simplifying assumptions:

Memory stall cycles

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

\[ \text{Instructions} / \text{Program} \times \text{Misses} / \text{Instruction} \times \text{Miss penalty} \]
**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)
  - $\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, L-cache miss rate = 5%
  - $\text{AMAT} = 1 + 0.05 \times 20 = 2\text{ns}$
    - 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

Directly mapped: 12 modulo 8 = 4
2-way set asso (4 sets): 12 modulo 4 = 0 anywhere in set 0
Fully association: anywhere
Spectrum of Associativity

For a cache with 8 entries

One-way set associative (direct mapped)

<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>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></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>
<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>
<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 *(tag, V bit not shown)*

<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>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[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]</td>
</tr>
</tbody>
</table>

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 52
Associativity Example

- 2-way set associative
  - Replacement: LRU
  - Block address | Cache index | Hit/miss | Cache content after access
    | Set 0 | Set 1 |
  - 0 | 0 | miss | Mem[0] |
  - 8 | 0 | miss | Mem[0] Mem[8] |
  - 0 | 0 | hit | Mem[0] Mem[8] |
  - 6 | 0 | miss | Mem[0] Mem[6] |

- Fully associative
  - Block address | Hit/miss | Cache content after access
    | Mem[0] |
  - 0 | miss | Mem[0] |
  - 8 | miss | Mem[0] Mem[8] |
  - 0 | hit | Mem[0] Mem[8] |
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

Address
31 30 \ldots 12 11 10 9 8 \ldots 3 2 1 0

Tag
Index

Each row Is a set

Four-way set associative: 4 blocks/set

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 55
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 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 = 400 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 analyse
  - Use system simulation
Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access
Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```c
for (int j = 0; j < n; ++j)
{
    double cij = C[i+j*n];
    for (int k = 0; k < n; k++)
        cij += A[i+k*n] * B[k+j*n];
    C[i+j*n] = cij;
}
```
DGEMM Access Pattern

- C, A, and B arrays

```plaintext
older accesses
new accesses
```

![Diagram showing access patterns for C, A, and B arrays]

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 64
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6    for (int j = sj; j < sj+BLOCKSIZE; ++j)
7    {
8      double cij = C[i+j*n];/* cij = C[i][j] */
9      for ( int k = sk; k < sk+BLOCKSIZE; k++ )
10         cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11      C[i+j*n] = cij; /* C[i][j] = cij */
12    }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17    for ( int si = 0; si < n; si += BLOCKSIZE )
18      for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19         do_block(n, si, sj, sk, A, B, C);
20 }
Blocked DGEMM Access Pattern
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