Chapter 7
Large and Fast: Exploiting Memory Hierarchy
Topics

- Memory types & categories
- Memory Hierarchy
- Caches
- Virtual Memory
After CPU design, memory is next.
Memory Components

Memory is Functionally a set of registers to hold programs & data
Locality Principle

- Foldoc

Temporal locality (Time):
  - Tendency to reuse recently accessed data
    => Keep most recent data items closer to the processor

Spatial locality (Space):
  - Tendency to reference data items that are close to other recently accessed data items
    => Move blocks of contiguous words to the upper level
Locality Principle

- We take advantage by implementing the memory of a computer as a memory hierarchy
- Move blocks of multiple contiguous words in memory to upper levels of hierarchy
- Accesses that miss, go to lower levels
- If the hit rate is high enough, we have an effective memory hierarchy
Fact: Program Code Have Locality

- From program structure
  - Most programs contain loops
    => Instructions & data are likely to be accessed repeatedly
    => Temporal locality
  - Elements of an array or record are stored together
    => We usually access the elements of related structure simultaneously
    => Spatial locality
Memory & Performance

- The concepts used to build memory system affects other aspects of a computer
  - How OS manages memory & I/O
  - How compilers generate code
  - How applications use the machine
Memories: Review

- **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)
  - Reading is destructive
  - very small but slower than SRAM (factor of 5 to 10)
  - For more details, see Appendix B, Section B.9
Memory Hierarchy

- Users want large and fast memories!
- In 2004:
  - **SRAM**
    - Access times: .5 - 5ns
    - Cost: $4000 - $10,000 per GB.
  - **DRAM**
    - Access times: 50 - 70ns
    - Cost: $100 - $200 per GB.
  - **Disk**
    - Access times: 5 - 20 million ns
    - Cost: $.50 - $2 per GB.
- Try and give it to them anyway
  - build a memory hierarchy
By implementing memory hierarchy, user has the illusion of large & fast memory.

<table>
<thead>
<tr>
<th>Current technology</th>
<th>Speed</th>
<th>Size</th>
<th>Cost $/bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache (SRAM)</td>
<td>Fastest</td>
<td>Smallest</td>
<td>Highest</td>
</tr>
<tr>
<td>Main (DRAM)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secondary (Magnetic)</td>
<td>Slowest</td>
<td>Biggest</td>
<td>Lowest</td>
</tr>
</tbody>
</table>
Memory Hierarchy

- Multiple levels of memory with different speeds & sizes
  - Fastest memory is the most expensive per bit
  - Fastest memory is the smallest in size
Memory Hierarchy

- Put faster memory closer to CPU

Goal:
- Present the user with as much memory as available
- Provide access at the speed offered by the fastest memory
- Data could be copied between adjacent levels only

Result:
- User thinks that memory is large
- Memory access is fast as fastest memory

Illusion
- CPU access time determined by level 1
- CPU has memory as large as level n
Memory Hierarchy

- Every pair of levels in the memory hierarchy can be thought of as having an upper & lower level.
- Within each level, block is the unit of information.
- Transfer is performed block-wise.
## Terminology

- **Block:**
  - Minimum unit of data that can be either present or not present in a memory level

- **Hit:**
  - When the requested data is found in the upper level

- **Hit rate (ratio):**
  - The fraction of memory accesses found in the upper level
  - Used as a measure of performance of the memory hierarchy
  - The larger the better

- **Hit time:**
  - Time to access the upper level of memory hierarchy
  - Includes time to determine whether an access is a hit or miss
Terminology

- **Miss:**
  - Data requested is not found in the upper level
  - Data need to be read from lower level & stored into upper level

- **Miss rate (ratio):**
  - 1 - hit ratio
  - The fraction of memory accesses not found in the upper level
  - The lower the better

- **Miss penalty:**
  - Time to
    - replace a block in the upper level with the corresponding block from lower level +
    - deliver this block to the processor
Cache

What is a cache?
- Storage managed to take advantage of locality principle
- Webster’s Universal College Dictionary, 1997

"1. A hiding place for ammunition, food, treasure, etc.
2. A place of computer HW or section of RAM dedicated to selectively storing and speeding access to frequently used program commands or data"
Cache

Example:

<table>
<thead>
<tr>
<th>X4</th>
<th>X1</th>
<th>Xn - 2</th>
<th>Xn - 1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
</table>

- Xn not in cache
- Referencing Xn => miss

Fetch Xn from memory and save into cache

Issues:

- How do we know if a data item is in the cache?
- If it is, how do we find it?
Direct-Mapped Cache

- Used up to 1992 MIPS CPU models
- Each memory location maps to exactly one location in the cache
- One-to-many mapping
- Formula used:
  (block address) modulo (Number of cache blocks in the cache)
- Part of the memory address is used as tag
- Several items at the lower level share locations in the upper level
Direct-Mapped Cache

- Example:
  - Block size is one word of data

- Problem:
  - If two different memory addresses that map to the same location are accessed interchangeably, they will pushing each other

- Possible solution:
  - Use 2 caches in parallel, looking up memory locations in both
Direct Mapped Cache

- Mapping:
  - Address is modulo the number of blocks in the cache
  - Cache has 8 entries
  - Cache index:
    - Use the rightmost 3-digit to map the address

- Cache accessed directly using low-order bits
Direct Mapped Cache

- Problem:
  - How do we know whether the requested word is in the cache?

- Solution: Tag field
  - Added to data word in cache
  - Contain address information to where the data belongs
  - Need only contain the upper portion of the address (not used as index to the cache)
Direct Mapped Cache

- **Problem:**
  - How do we know that the block has the valid information?

- **Solution: Valid bit**
  - Added to data word in cache
  - 1 means data is valid (valid address)
  - 0 means data is invalid (Invalid address)
Direct Mapped Cache

- **Example:**
  - Cache has 8 blocks
  - 1 block = 1 word (4 bytes)
  - 32-bit address
  - Byte offset:
    - The first 2 bits give the byte number (byte offset)
  - Cache index:
    - Next Low-order 3 bits of the address will give the block number (cache index)
  - The rest of the bits give the tag (address of block in higher level memory)
  - Both data & tag are stored in each entry of the cache
  - Some other bits can be stored
    - Valid bit
    - ...

**Diagram:**
- Address (showing bit positions)
- Hit
- Tag
- Valid
- Data
- Index
- Byte offset
- 32-bit address
Direct Mapped Cache

Example
- Draw the cache after inserting the following blocks:
  10110, 11010, 10000, 00011, 10010

Initial Cache

<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>
Direct Mapped Cache

- After 10110

- After 11010
Direct Mapped Cache

- After 10000

- After 00011

<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>Memory(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>Memory(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>Memory(10110)</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></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>Memory(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>Memory(11010)</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Memory(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>Memory(10110)</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache

- After 10010

<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>Memory(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>Memory(10010)</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Memory(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>Memory(10110)</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache for MIPS

- Cache size $2^{10} = 1024$
- Index bits = 10
- Tag = $32 - 10 - 2 = 20$
- tag from cache compared against upper portion of address
- If match Þ Entry in cache corresponds to requested address
- If valid bit = 1 Þ Request hits Þ word supplied to processor
- If valid bit = 0 Þ Request misses Þ bring word into cache

Address (showing bit positions)

31 30 13 12 11  2  1  0

Valid  Tag  Data

Index

0
1
2

- - -
- - -
1021
1022
1023

20
32

Hit

Data
Direct Mapped Cache

Exercise:
- How many bits are needed for 64 KB of data

Assumptions:
- 1-word block
- 32-bit address

Valid | Tag | Data
--- | --- | ---
1 bit + (32-14-2) = 16 + 32 => total 49 bits / entry
Direct Mapped Cache

Answer:
- Formulas:
  - Total number of bits needed = Cache entry size * block size
  - Size of cache entry = Data size + Tag size + Valid bit size
- Cache has 64KB of data ( = 2^{14} words)
- One-word blocks => 2^{14} blocks
- Total bits needed = 2^{14} \times 49 = 2^{10} \times 2^4 \times 49
  = 2^{10} \times 16 \times 49 = 784 \times 2^{10}
  = 784 K-Bits

More Exercises: See examples, pages 479 & 480
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 data only into the cache (write-back the cache later)

- **Write misses:**
  - Read the entire block into the cache, then write the word
Handling Cache Misses

- The action taken for cache miss depends on whether the action is to access instruction or data

  **Data Access**
  - Stall the processor until the memory responds with the data

  **Instruction access:**
  - The contents of the instruction register are invalid
    - Get the instruction from the lower-level memory into the cache
    - Decrement PC contents by 4 to get the address of the current instruction, since the PC is already incremented at the first clock cycle
Handling Cache Misses

Instruction Access Steps:
1. Send the original PC value (PC-4) to the memory
2. Instruct main memory to perform read & wait for the memory to complete its access
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address in the tag field, and turning the valid bit on
4. Restart the instruction execution at the first step, which will re-fetch the instruction, this time finding it in the case
Handling Write Misses

- Three approaches
  - Write through
  - Write buffer
  - Write back
Write-Through Approach

**Idea:**
- Always write data into both memory & cache
- Or better, just write to cache, updating both tag & data
- Or use cache buffer then write to memory

**Steps:**
1. Index the cache (Bits 15-2 of address)
2. Write the tag, data, & valid bit into cache
3. Write the word to main memory using the entire address

**Advantages**
- Simple algorithm

**Disadvantages**
- Poor performance
  - If buffer is full, CPU stalls
  - Writing into main memory slows down the machine
Write-Buffer Approach

- **Idea:**
  - A write buffer (queue) stores the data while it is waiting to be written to memory.
  - If the write buffer is full, processor must stall until there is an empty position in the write buffer.

- **Advantages**
  - Better than write-through.

- **Disadvantages**
  - Stalls might still occur.
Write-Back Approach

- Handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

- Steps:
  1. Index the cache
  2. Write the tag, data, & valid bit into cache
  3. The modified block is written to the memory only when it is replaced

- Advantages:
  - Improves performance

- Disadvantages:
  - Complex algorithm
Multiword Cache Blocks

- Taking advantage of spatial locality
  - Have a cache block larger than one word
  - A MUX is needed to select the right word in the block
  - An extra block offset field is used to control the MUX
  - Same mapping can be used to find a block in the cache
- Caches with multiword blocks use memory more efficiently
Multiword Cache Blocks

Address (Showing bit position)

- 31 16 15 4 3 2 1 0
- Valid bit
- Hit
- Tag
- Index

- 16 bits Tag
- 128 bits Data

- Block offset
- Data
- 4K entries

- MUX
- 32
- 32
- 32
- 32
- 32

- 16
- 32
- 32
- 32
- 32

- Offset
- Byte
Multiword Cache Blocks Example

- Intrinsity FastMATH Processor (Details, pp. 485-487)
Multiword Cache Blocks

Exercise:

Given:
- Cache with 64 blocks
- Block size = 16-bytes

Required:
- What block number does byte address 1200 map to?
Multiword Cache Blocks

Answer:

Block address = byte address / bytes per block

Block # = (Block address) mod (# cache blocks)

= (Byte address/Bytes per block) mod (# cache blocks)

= (1200 / 16) mod (64)

= 75 mod 64

= 11
Handling Multiword Block Misses

- Read hits
  - This is what we want, proceed

- Read misses
  - Stall the CPU
  - Fetch the entire block from memory
  - Deliver block to cache
  - Restart
Handling Multiword Block Misses

- **Write hits**
  - We can’t just write the tag & data
    - Have to check if the other words are from the same block in memory
  - What shall we do?
    - First write the data & set the tag of the word
    - Compare the tags of all the words in that block
    - If tags are equal, proceed
    - If tags are not equal we have a write miss

- **Write misses**
  - Read the entire block into the cache, then write the word
Memory Systems Supporting Cache

Three organizational options

- One-word-wide memory organization
  - All accesses are made sequentially

- Wide memory organization
  - Increase the bandwidth by widening the memory and the busses between the processor and memory
  - Allows parallel access to all words of the block
  - Decrease both access time and transfer time
  - Cost of multiplexor is added

- Interleaved memory organization
  - Widening the memory but not the interconnection bus
  - Memory organized in banks to read / write multiple words in one access time
  - Each bank could be one word wide
  - Each bank can write independently
  - Still pay a cost to transmit each word
Memory System Supporting Cache

- Make reading multiple words easier by using banks of memory
- It can get a lot more complicated...

One-word-wide organization

Wide memory organization
Memory, buffer, & cache are wide

Interleaved organization
Narrow bus & cache,
Interleaved memory
Performance

- Increasing the block size tends to decrease miss rate
- Use split caches because there is more spatial locality in code

<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>gcc</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>spice</td>
<td>4</td>
<td>0.3%</td>
<td>0.6%</td>
<td>0.4%</td>
</tr>
</tbody>
</table>
Performance

- Two ways of improving performance:
  - Decreasing the miss ratio
    - Reducing the probability that two different memory blocks will contend the same cache location
  - Decreasing the miss penalty
    - Multi-level caching:
      - Add additional level of hierarchy
      - Common on desktop computers for less than $1000!
Decreasing Miss Ratio with Associativity

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>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
Associative Caches

- Fully Associative
  - A block can be placed in any location in the cache
  - Entries are searched to find a given block
  - Search should be done in parallel using a comparators associated with each entry
  - Too expensive

- Set-Associative
  - Combined direct-mapped & fully associative
  - Has a fixed number of locations (at least 2) where each block can be placed
  - All blocks in the set are searched for a match

- n-way-set associative cache
  - Each block maps to a unique set
  - The block can be placed in any element in the set

- Set containing a memory block
  \[ \text{Set containing a memory block} = \text{(block#)} \mod \text{(\# sets in the cache)} \]

- See Figs. 7.13, p. 498
Associative Caches

- An implementation of 4-way set-associative cache
  - 4 comparators & 4x1 MUX needed
Set-Associative Caches

Performance

<table>
<thead>
<tr>
<th>Associativity</th>
<th>1 KB</th>
<th>2 KB</th>
<th>4 KB</th>
<th>8 KB</th>
<th>16 KB</th>
<th>32 KB</th>
<th>64 KB</th>
<th>128 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-way</td>
<td>12%</td>
<td>11%</td>
<td>10%</td>
<td>9%</td>
<td>8%</td>
<td>7%</td>
<td>6%</td>
<td>5%</td>
</tr>
<tr>
<td>Two-way</td>
<td>11%</td>
<td>9%</td>
<td>7%</td>
<td>5%</td>
<td>4%</td>
<td>3%</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>Four-way</td>
<td>10%</td>
<td>8%</td>
<td>6%</td>
<td>4%</td>
<td>3%</td>
<td>2%</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Eight-way</td>
<td>9%</td>
<td>7%</td>
<td>5%</td>
<td>3%</td>
<td>2%</td>
<td>1%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing performance of set-associative caches](image-url)
Decreasing Miss Penalty With Multilevel Caches

- Add a second level cache:
  - Often primary cache is on the same chip as the processor
  - Use SRAMs to add another cache above primary memory (DRAM)
  - Miss penalty goes down if data is in 2nd level cache

- Example:
  - CPI of 1.0 on a 5 Ghz machine with a 5% miss rate, 100ns DRAM access
  - Adding 2nd level cache with 5ns access time decreases miss rate to .5%
Cache Complexities

Not always easy to understand implications of caches:

Theoretical behavior of Radix sort vs. Quicksort

Observed behavior of Radix sort vs. Quicksort
Cache Complexities

Here is why:

Memory system performance is often critical factor
- multilevel caches, pipelined processors, make it harder to predict outcomes
- Compiler optimizations to increase locality sometimes hurt ILP

Difficult to predict best algorithm: need experimental data
Virtual Memory

- Similar to caching, but between main memory & secondary storage instead of RAM & CPU
- Main memory acts as a “cache” for the secondary storage
- Only a portion of the program can be active in main memory, the rest will be in secondary storage
- Multiple program can share the same memory
- Programs sharing the memory change dynamically
Virtual Memory

Advantages
- Allows efficient & safe sharing of memory among multiple programs
- Larger address space (Illusion of having more physical memory) without extra programming requirement

Disadvantages
- Overhead for
  - memory management
  - handling page faults
  - program relocation (converting virtual address space into physical address)
- Programs need to be protected from each other
Virtual Memory

Protection

- A set of mechanisms for ensuring multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or un-intentionally, with one another by reading/writing each other's data. They also isolate the OS from user processes.
Virtual Memory

Virtual addresses

Address translation

Physical addresses

Disk addresses
Virtual Memory

Address Space

- The number of addressable words in a program
  - Depends on the number of bits in the address field, not on the number of memory word actually available
- From [http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?address+space](http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?address+space)
  - The range of addresses which a processor or process can access, or at which a device can be accessed. The term may refer to either physical address or virtual address.

Physical address: ([http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?physical+address](http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?physical+address))

- The address presented to a computer's main memory in a virtual memory system, in contrast to the virtual address which is the address generated by the CPU. A memory management unit translates virtual addresses into physical addresses.
Virtual Memory

- Virtual address (Program’s address space)
  - An address that corresponds to a location in virtual space
  - Translated by address mapping to a physical address when memory is accessed
  - A separate range of memory locations accessible only to a specific program. Each program should have its own address space
- FOLDOC:
  - A memory location accessed by an application program in a system with virtual memory such that intervening hardware and/or software maps the virtual address to real (physical) memory. During the course of execution of an application, the same virtual address may be mapped to many different physical addresses as data and programs are paged out and paged in to other locations.
Virtual Memory

- Address translation (mapping):
  - The process by which a virtual address is mapped to an address used to access memory

Virtual address

31 30 29 28 27 15 14 13 12 11 10 9 8 3 2 1 0

Virtual page number | Page offset

Translation

Physical address

29 28 27 15 14 13 12 11 10 9 8 3 2 1 0

Physical page number | Page offset
Virtual Memory

- Segment:
  - *(Foldoc)*
    - A separately relocatable section of an executable program.
    - A collection of *pages* in a *memory management* system.

- Segmentation:
  - A variable-size address mapping scheme in which an address consists of 2 parts
    - Segment number
    - Segment offset
  - Segment number is mapped into physical address
Virtual Memory

- Overlays
  - Divide program into mutually exclusive segments that could fit into memory
  - Each segment contain both program and data
  - Size of each segment could be different
  - Problems:
    - The responsibility of the programmer to divide & control loading & unloading of overlays
    - Leads to fragmentation
    - If tow modules mutually need each other => Thrashing
Virtual Memory

Virtual memory:
- Intended to relieve the programmer from managing overlays
- Virtual address space is broken into a number of equal-sized pages
- Physical address space is also broken into the same page size (page frame)
- Two or more pages can occupy the same page in physical address space
- Consecutive pages in virtual address space need not be consecutive in physical address space
Virtual Memory

- Page
  - VM block
  - Usual size
    - 512-64KBytes
    - Modern computer size up to 4 MBytes

- Page fault
  - An event that occurs when an accessed page is not present in memory
  - Page is not in memory (~Virtual memory miss)
Address Translation

Example:

- Page size = $2^{12} = 4$ KB
  - Dictates the maximum page offset
- Number of physical pages allowed = $2^{18}$
- Physical memory size = $2^{12} \times 2^{18} = 2^{30} = 1$ GB
- Virtual address space = $2^{32} = 4$ GB
- Virtual Address = Virtual page number + page offset
- Physical Address = Physical page number + page offset

⇒ Need to translate Virtual to Physical page number

- Translation is performed using tables
- Page offset stays the same for both virtual & physical addresses
Address Translation

Example:
Translating Virtual to Physical page number:

Virtual address

31 30 29 28 27 15 14 13 12 11 10 9 8 3 2 1 0

<table>
<thead>
<tr>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 28 27</td>
<td></td>
</tr>
</tbody>
</table>

Translation

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8</td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Physical address
Pages Faults

- Occurs when the valid bit for a virtual page is off
- Data is not in memory and needs to be retrieved from disk
- OS must be given control through exception mechanism
  - OS finds the page in the next level of the hierarchy & decide where to place the page in the memory
  - OS creates a data structure to track which process & which virtual address use each physical page
  - OS must choose which page to replace
  - Replaced page written to swap space on the disk
Pages Faults

- Huge miss penalty
  - High access time
  - Pages should be fairly large (e.g. 64 KB for new systems)
- Reducing page faults is important
  - Replacement strategies needed
- Faults can be handled in software or hardware
  - Software have clever algorithms to choose how to place pages
Replacement strategies

- Least-recently used (LRU):
  - The block replaced is the one that has been unused for the longest time
  - OS keeps track of which page wasn’t recently used with “Use (reference) bit”
  - OS periodically clears the reference bit & later records them to determine which pages were changed during a specific time period

- First-in-first-out (FIFO)
  - The block replaced is the one that is in the memory the longest time
  - Not practical

Methods of replacement

- Write-through
  - Too expensive
  - Write takes too long

- Use write back
  - Only if the page is removed from main memory, write it back
Page Placement

- Fully associative memory
  - A cache structure in which a block can be replaced in any location in the memory
  - Address is used as a tag
  - Tags must be compared simultaneously (associatively) with requested address
  - If address matches tag, its associated data is accessed

- Fully associative page replacement
  - Placing the page
  - Finding a page again
Page Placement

- The operating system uses sophisticated algorithms & complex data structures to track page usage
- Goal:
  - Reduces page fault rate
- Method:
  - Locate page by using a page table
Page Table (PT)

- Used to map virtual address space of a program into physical address space in main memory
- Resides in memory
- Not a cache => No tag field required
- Contains physical page numbers of the program
- May contain entries for pages not in memory
- Indexed with page number (an index for each virtual page)
- Each program has its own page table
Page Table

Virtual page number

Page table
Valid
Physical page or disk address

Physical memory

Disk storage
**Page Table**

- **Valid bit:**
  - Used the same way it is used in cache
  - Valid bit = 0
    - Page not present in memory
    - If page is addressed, page fault occurs
  - Valid bit = 1
    - Entry contains physical page number

- **Page table register:**
  - Used to point to the page table
  - When a program is downloaded the address of its page table should be loaded into the page table register
Page Table

Example:
- Virtual address space = $2^{32}$
- Physical memory size = $2^{30}$
- Page size = $2^{12}$
- Number of page table entries = $2^{32} - 2^{12} = 2^{20}$
- Number of addressable pages in physical memory = $2^{30} - 2^{12} = 2^{18}$
- # bits in each table entry = 18 (page#) + (1 valid bit) = 19-bits wide
Page Table

Block diagram

Virtual address

Virtual page number

Page offset

Valid

Physical page number

Page table
(Size = 4KB)

If 0 then page is not present in memory

Physical page number

Page offset

Physical address

If 0 then page is not present in memory

Physical page number

Page offset

Physical address

Page table register

Virtual address

Virtual page number

Page offset
Page Table

Problems:

- Every access requires two steps:
  - Memory access to obtain physical address
    - If necessary handle page fault
  - Memory access to obtain data
- PT can be very large
- Several PTs needed for several programs
- All must reside in memory
- Too much memory for PTs, where do we put the programs?
  => Need to minimize memory space used by PTs
Page Table

Solutions:

- To improve time performance, use locality principle
  - Translation of a virtual page number will most probably be needed again
  - Use a cache to keep track of recently-used translation (Translation look-aside buffer)

- To improve size, use one of the following
  - Start with a small PT, increase when necessary
  - Use the idea used for stack and heap:
    - Put 2 PTs in one area and let them grow in opposite directions
  - Apply hashing function to virtual address
    - Lookup will be more complicated
  - Use multiple-level PTs
    - PT should be paged in the same manner as virtual memory
Page Table

- State of a program (process):
  - The page table, program counter, & registers specify the state of a program
  - If we want to allow another process to use the processor, we must save this state

- Active process:
  - If it is in possession of the processor

- Inactive process:
  - If the process does not possess the processor

- Swap space:
  - Space on the disk reserved by the OS for all pages of a program when its process is created (for the full virtual memory space of a process)
Page Table

Since the page table is stored in main memory, every memory access by a program can take double the time:
- One memory access to get the physical address
- Another access to get the data

According to the (spatial & temporal) locality principle, the virtual page number might be needed again in the near future.

Modern processors keep a special cache to keep track of recently-used translations, traditionally called “Translation Look-aside Buffer (TLB)”
Translation Lookaside Buffer (TLB)

- A cache that keeps track of recently used address mappings
- Used to avoid an access to the page table
- Contains a subset of virtual-to-physical page mappings that are in the PT
- Because TLB is a cache, it must have a tag field
- If there is no matching entry in TLB for a page, the PT must be examined to supply
  - Physical page number for the page to be used in building the TLB
  - or Indicate that the page resides on disk, in which case a page fault occurs
Translation Lookaside Buffer

Virtual page number

TLB

Physical memory

Page table

Disk storage

Physical page address

Valid Dirty Ref

Tag

Physical page address

Valid Dirty Ref

or disk address

Physical page address

Virtual page number

Valid Dirty Ref

Physical page address

Valid Dirty Ref

or disk address
Translation Lookaside Buffer

- Typical values
  - TLB size: 16-512 entries
  - Block size: 1-2 PT entries (4-8 bytes each)
  - Hit time: 0.5 - 1 clock cycle
  - Miss penalty: 10 - 100 clock cycles
  - Miss rate: 0.01 - 1%
Translation Look-aside Buffer

Making Address Translation Fast

- Handling “read”
  - Every tag is compared against the index value
  - If valid bit of matching entry = 1, access is a “hit”
  - ⇒ Cache address = physical page # + offset
Translation Look-aside Buffer (TLB)

- TLB for Intrinsity
- FastMATH
Translation Look-aside Buffer (TLB)

- Processing read/write-through for Intrinsity FastMATH TLB & cache

```
Virtual address

TLB access

TLB access

TLB hit?

TLB miss exception

Yes

Physical address

No

Try to read data from cache

Cache miss stall while read block

Cache hit?

No

Write?

Yes

Try to write data to cache

Cache miss stall while read block

Cache hit?

No

Write access bit on?

Write protection exception

No

Yes

Yes

Write data into cache, update the dirty bit, and put the data and the address into the write buffer.

Deliver data to the CPU

Yes

No
```
Possible Event Combinations

- Three types of misses
  - TLB miss
  - Page fault
  - Cache miss

- All the combinations of these three events should be considered
### Possible Event Combinations

<table>
<thead>
<tr>
<th>Cache</th>
<th>TLB</th>
<th>VM</th>
<th>Possible? Under what circumstances</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>hit</td>
<td>hit</td>
<td>Possible. But PT is never checked if TLB hits</td>
</tr>
<tr>
<td>hit</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, entry found in PT; after retry data found in cache</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>hit</td>
<td>TLB misses, entry found in PT; after retry, data misses in cache</td>
</tr>
<tr>
<td>miss</td>
<td>miss</td>
<td>miss</td>
<td>(Worst-case) TLB misses &amp; followed by page fault; after retry, data must miss in cache</td>
</tr>
<tr>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td>Impossible; can’t have a translation in TLB if page not present in memory</td>
</tr>
<tr>
<td>hit</td>
<td>hit</td>
<td>miss</td>
<td>Impossible; can’t have a translation in TLB if page not present in memory</td>
</tr>
<tr>
<td>hit</td>
<td>miss</td>
<td>miss</td>
<td>Impossible; data can’t be allowed in cache if the page is not present in memory</td>
</tr>
</tbody>
</table>
Modern Systems

Real Stuff: Pentium P4 & AMD

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Pentium P4</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address</td>
<td>32 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Physical address</td>
<td>36 bits</td>
<td>40 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>4 KB, 2/4 MB</td>
<td>4 KB, 2/4 MB</td>
</tr>
<tr>
<td>TLB organization</td>
<td>1 TLB for Instructions and 1 TLB for data</td>
<td>2 TLBs for Instructions and 2 TLBs for data</td>
</tr>
<tr>
<td></td>
<td>Both are four-way set associative</td>
<td>Both L1 TLBs fully associative, LRU replacement</td>
</tr>
<tr>
<td></td>
<td>Both use pseudo-LRU replacement</td>
<td>Both L2 TLBs are four-way set associativity, round-robin LRU</td>
</tr>
<tr>
<td></td>
<td>Both have 128 entries</td>
<td>Both L1 TLBs have 40 entries</td>
</tr>
<tr>
<td></td>
<td>TLB misses handled in hardware</td>
<td>Both L2 TLBs have 512 entries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB misses handled in hardware</td>
</tr>
</tbody>
</table>

**FIGURE 7.34** Address translation and TLB hardware for the Intel Pentium P4 and AMD Opteron. The word size sets the maximum size of the virtual address, but a processor need not use all bits. The physical address size is independent of word size. The P4 has one TLB for instructions and a separate identical TLB for data, while the Opteron has both an L1 TLB and an L2 TLB for instructions and identical L1 and L2 TLBs for data. Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.
Modern Systems

Real Stuff: Pentium P4 & AMD

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Intel Pentium P4</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache organization</td>
<td>Split Instruction and data caches</td>
<td>Split Instruction and data caches</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>8 KB for data, 96 KB trace cache for RISC Instructions (12K RISC operations)</td>
<td>64 KB each for Instructions/data</td>
</tr>
<tr>
<td>L1 cache associativity</td>
<td>4-way set associative</td>
<td>2-way set associative</td>
</tr>
<tr>
<td>L1 replacement</td>
<td>Approximated LRU replacement</td>
<td>LRU replacement</td>
</tr>
<tr>
<td>L1 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 write policy</td>
<td>Write-through</td>
<td>Write-back</td>
</tr>
<tr>
<td>L2 cache organization</td>
<td>Unified (Instruction and data)</td>
<td>Unified (Instruction and data)</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>512 KB</td>
<td>1024 KB (1 MB)</td>
</tr>
<tr>
<td>L2 cache associativity</td>
<td>8-way set associative</td>
<td>16-way set associative</td>
</tr>
<tr>
<td>L2 replacement</td>
<td>Approximated LRU replacement</td>
<td>Approximated LRU replacement</td>
</tr>
<tr>
<td>L2 block size</td>
<td>128 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 write policy</td>
<td>Write-back</td>
<td>Write-back</td>
</tr>
</tbody>
</table>

**FIGURE 7.35** First-level and second-level caches in the Intel Pentium P4 and AMD Opteron. The primary caches in the P4 are physically indexed and tagged; for a discussion of the alternatives, see the Elaboration on page 527.
Modern Systems

- Things are getting complicated!

<table>
<thead>
<tr>
<th>MPU</th>
<th>AMD Opteron</th>
<th>Intrinsity FastMATH</th>
<th>Intel Pentium 4</th>
<th>Intel PXA250</th>
<th>Sun UltraSPARC IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction set architecture</td>
<td>IA-32, AMD64</td>
<td>MIPS32</td>
<td>IA-32</td>
<td>ARM</td>
<td>SPARC v9</td>
</tr>
<tr>
<td>Intended application</td>
<td>server</td>
<td>embedded</td>
<td>desktop</td>
<td>low-power</td>
<td>server</td>
</tr>
<tr>
<td>Die size (mm²) (2004)</td>
<td>193</td>
<td>122</td>
<td>217</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>Instructions issued/clock</td>
<td>3</td>
<td>2</td>
<td>3 RISC ops</td>
<td>1</td>
<td>4 × 2</td>
</tr>
<tr>
<td>Clock rate (2004)</td>
<td>2.0 GHz</td>
<td>2.0 GHz</td>
<td>3.2 GHz</td>
<td>0.4 GHz</td>
<td>1.2 GHz</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>64 KB, 2-way set associative</td>
<td>1.6 KB, direct mapped</td>
<td>12000 RISC op trace cache (~96 KB)</td>
<td>32 KB, 32-way set associative</td>
<td>32 KB, 4-way set associative</td>
</tr>
<tr>
<td>Latency (clocks)</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Data cache</td>
<td>64 KB, 2-way set associative</td>
<td>1.6 KB, 1-way set associative</td>
<td>8 KB, 4-way set associative</td>
<td>32 KB, 32-way set associative</td>
<td>64 KB, 4-way set associative</td>
</tr>
<tr>
<td>Latency (clocks)</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>TLB entries (I/D/L2 TLB)</td>
<td>40/40/512</td>
<td>16</td>
<td>128/128</td>
<td>32/32</td>
<td>128/512</td>
</tr>
<tr>
<td>Minimum page size</td>
<td>4 KB</td>
<td>4 KB</td>
<td>4 KB</td>
<td>1 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>On-chip L2 cache</td>
<td>1024 KB, 16-way set associative</td>
<td>1024 KB, 4-way set associative</td>
<td>512 KB, 8-way set associative</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Off-chip L2 cache</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>16 MB, 2-way set associative</td>
</tr>
<tr>
<td>Block size (L1/L2, bytes)</td>
<td>64</td>
<td>64</td>
<td>64/128</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

**FIGURE 7.36 Desktop, embedded, and server microprocessors in 2004.** From a memory hierarchy perspective, the primary differences between categories is the L2 cache. There is no L2 cache for the low-power embedded, a large on-chip L2 for the embedded and desktop, and 16 MB off chip for the server. The processor clock rates also vary: 0.4 GHz for low-power embedded, 1 GHz or higher for the rest. Note that UltraSPARC IV has two processors on the chip.
Some Issues

- Processor speeds continue to increase very fast
Conclusion

- Processor speeds continue to increase very fast
  - much faster than either DRAM or disk access times
- Design challenge:
  - Dealing with this growing differences
- Trends:
  - Synchronous SRAMs (provide a burst of data)
  - Redesign DRAM chips to provide higher bandwidth or processing
  - Restructure code to increase locality
  - Use pre-fetching (make cache visible to ISA)
Homework!

See Chapter 7 Assignment