ECE7995
(6) Improving Cache Performance

[Adapted from Mary Jane Irwin’s slides (PSU)]
**Measuring Cache Performance**

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then
  
  \[ \text{CPU time} = IC \times CPI \times CC \]
  
  \[ = IC \times (\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles}) \times CC \]

  - IC: instruction count, CPI: cycles per instruction; CC: clock cycle time

- Memory-stall cycles come from cache misses
  
  \[ \text{Memory-stall cycles} = \text{data accesses/program} \times \text{miss rate} \times \text{miss penalty} \]

- Average Memory Access time = Hit Rate x Hit Time + Miss Rate x Miss Penalty
Review: The “Memory Wall”

- Logic vs DRAM speed gap continues to grow
Impacts of Cache Performance

- Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI)
  - The memory speed is unlikely to improve as fast as processor cycle time. When calculating $\text{CPI}_{\text{stall}}$, the cache miss penalty is measured in processor clock cycles needed to handle a miss.
  - The lower the $\text{CPI}_{\text{ideal}}$, the more pronounced the impact of stalls.

- A processor with a $\text{CPI}_{\text{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr’s, and 2% I$ and 4% D$ miss rates:

  Memory-stall cycles = 2% × 100 + 36% × 4% × 100 = 3.44
  So $\text{CPI}_{\text{stalls}} = 2 + 3.44 = 5.44$

- What if the $\text{CPI}_{\text{ideal}}$ is reduced to 1? 0.5? 0.25?

- What if the processor clock rate is doubled (doubling the miss penalty)? For miss penalty of 200, memory stall cycles = 2% × 200 + 36% × 4% × 200 = 6.88
Reducing Cache Miss Rates #1

1. Allow more flexible block placement

- In a **direct mapped cache** a memory block maps to exactly one cache block.
- At the other extreme, could allow a memory block to be mapped to any cache block – **fully associative cache**.

- A compromise is to divide the cache into **sets** each of which consists of \( n \) “ways” (**n-way set associative**). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are \( n \) choices)

  \[
  \text{(block address)} \mod (\# \text{ sets in the cache})
  \]
Set Associative Cache Example

Q1: How do we find it?

Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache).

Q2: Is it there?

Compare all the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache.

Two low order bits define the byte in the word (32-b words)

One word blocks
Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

8 requests, 2 misses

Try another access string: 0 1 2 3 0 8 11 0 3.

Solves the ping pong effect in a direct mapped cache due to **conflict** misses since now two memory locations that map into the same cache set can co-exist!
Four-Way Set Associative Cache

- $2^8 = 256$ sets each with four ways (each with one block)

Total bits required for the implementation:

$$256 \times 4 \times (\text{Data} + \text{tag} + \text{Valid})$$

$$= 256 \times 4 \times (32 + 22 + 1)$$
Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
<th>Byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for tag compare</td>
<td>Selects the set</td>
<td>Selects the word in the block</td>
<td></td>
</tr>
</tbody>
</table>

Decreasing associativity
- Direct mapped (only one way)
- Smaller tags
  - (the # of bits indicates the cache size in words)

Increasing associativity
- Fully associative
  - (only one set)
  - Tag is all the bits except block and byte offset
Costs of Set Associative Caches

- When a miss occurs, which way’s block do we pick for replacement?
  - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way’s block was used relative to the other blocks in the set
    - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way’s bit)

- N-way set associative cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available after set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available before the Hit/Miss decision
    - So it’s possible to just assume a hit and continue and recover later if it was a miss (remember that cache hit ratio is usually larger than 95%)
Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation.

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate).

Data from Hennessy & Patterson, *Computer Architecture*, 2003
Reducing Cache Miss Rates #2

2. Use multiple levels of caches

- With advancing technology has more than enough room on the die for bigger L1 caches or for a second level of caches – normally a unified L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache

- For our example, $CPI_{ideal}$ of 2, 100 cycle miss penalty (to main memory), 36% load/stores, a 2% (4%) L1I$ (D$) miss rate, add a UL2$ that has a 25 cycle miss penalty and a 0.5% miss rate (over program’s I & D accesses)

$$CPI_{stalls} = 2 + 0.02 \times 25 + 0.36 \times 0.04 \times 25 + 0.005 \times 100 + 0.36 \times 0.005 \times 100 = 3.54$$

(as compared to 5.44 with no L2$)
Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle
    - Smaller with smaller block sizes, small-associativity
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times
    - Larger with larger block sizes

- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate

- For the L2 cache, hit time is less important than miss rate
  - The L2’s hit time determines L1’s miss penalty
  - L2 local miss rate >> the global miss rate
# Key Cache Design Parameters

<table>
<thead>
<tr>
<th></th>
<th>L1 typical</th>
<th>L2 typical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total size (blocks)</td>
<td>250 to 2000</td>
<td>4000 to 250,000</td>
</tr>
<tr>
<td>Total size (KB)</td>
<td>16 to 64</td>
<td>500 to 8000</td>
</tr>
<tr>
<td>Block size (B)</td>
<td>32 to 64</td>
<td>32 to 128</td>
</tr>
<tr>
<td>Miss penalty (clocks)</td>
<td>10 to 25</td>
<td>100 to 1000</td>
</tr>
<tr>
<td>Miss rates (global for L2)</td>
<td>2% to 5%</td>
<td>0.1% to 2%</td>
</tr>
</tbody>
</table>
## Two Machines’ Cache Parameters

<table>
<thead>
<tr>
<th></th>
<th>Intel P4</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 organization</td>
<td>Split I$ and D$</td>
<td>Split I$ and D$</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>8KB for D$, 96KB for trace cache (~I$)</td>
<td>64KB for each of I$ and D$</td>
</tr>
<tr>
<td>L1 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 associativity</td>
<td>4-way set assoc.</td>
<td>2-way set assoc.</td>
</tr>
<tr>
<td>L1 replacement</td>
<td>~ LRU</td>
<td>LRU</td>
</tr>
<tr>
<td>L1 write policy</td>
<td>write-through</td>
<td>write-back</td>
</tr>
<tr>
<td>L2 organization</td>
<td>Unified</td>
<td>Unified</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>512KB</td>
<td>1024KB (1MB)</td>
</tr>
<tr>
<td>L2 block size</td>
<td>128 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 associativity</td>
<td>8-way set assoc.</td>
<td>16-way set assoc.</td>
</tr>
<tr>
<td>L2 replacement</td>
<td>~LRU</td>
<td>~LRU</td>
</tr>
<tr>
<td>L2 write policy</td>
<td>write-back</td>
<td>write-back</td>
</tr>
</tbody>
</table>
4 Questions for the Memory Hierarchy

- Q1: Where can a block be placed in the upper level?  
  *(Block placement)*

- Q2: How is a block found if it is in the upper level?  
  *(Block identification)*

- Q3: Which block should be replaced on a miss?  
  *(Block replacement)*

- Q4: What happens on a write?  
  *(Write strategy)*
**Q1&Q2: Where can a block be placed/found?**

<table>
<thead>
<tr>
<th></th>
<th># of sets</th>
<th>Blocks per set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td># of blocks in cache</td>
<td>1</td>
</tr>
<tr>
<td>Set associative</td>
<td>(# of blocks in cache)/associativity</td>
<td>Associativity (typically 2 to 16)</td>
</tr>
<tr>
<td>Fully associative</td>
<td>1</td>
<td># of blocks in cache</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Location method</th>
<th># of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>Set associative</td>
<td>Index the set; compare set’s tags</td>
<td>Degree of associativity</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Compare all blocks tags</td>
<td># of blocks</td>
</tr>
</tbody>
</table>
Q3: Which block should be replaced on a miss?

- Easy for direct mapped – only one choice
- Set associative or fully associative
  - Random
  - LRU (Least Recently Used)

- For a 2-way set associative cache, random replacement has a miss rate about 1.1 times higher than LRU.
- LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly
Q4: What happens on a write?

- **Write-through** – The information is written to both the block in the cache and to the block in the next lower level of the memory hierarchy
  - Write-through is always combined with a write buffer so write waits to lower level memory can be eliminated (as long as the write buffer doesn’t fill)

- **Write-back** – The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - Need a dirty bit to keep track of whether the block is clean or dirty

- Pros and cons of each?
  - Write-through: read misses don’t result in writes (so are simpler and cheaper)
  - Write-back: repeated writes require only one write to lower level
Improving Cache Performance

1. Reduce the time to hit in the cache
   - smaller cache
   - direct mapped cache
   - smaller blocks

2. Reduce the miss rate
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks
Improving Cache Performance (cont’d)

3. Reduce the miss penalty
   - smaller blocks
   - use a write buffer to hold dirty blocks being replaced so don’t have to wait for the write to complete before reading
   - check write buffer (and/or victim cache) on read miss – may get lucky
   - for large blocks fetch critical word first
   - use multiple cache levels – L2 cache not tied to CPU clock rate
   - faster backing store/improved memory bandwidth
     - wider buses
     - memory interleaving, page mode DRAMs
Summary: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back

- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost

- Simplicity often wins
Practice Questions:

Here is a series of an address references given as word address: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, and 11. Assuming a direct-mapped cache with 16 one-word blocks that is initially empty, label each reference in the list as a hit or miss and show the final contents of the cache.


<table>
<thead>
<tr>
<th>Cache set</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>48</td>
</tr>
<tr>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>21</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>27</td>
</tr>
<tr>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>
Practice Questions:

- Using the same series of references: 2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, and 11, show the hits and misses and final cache contents for a directly-mapped cache with four-word blocks and a total size of 16 words.


<table>
<thead>
<tr>
<th>Cache set</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[0, 1, 2, 3]</td>
</tr>
<tr>
<td>01</td>
<td>[4, 5, 6, 7]</td>
</tr>
<tr>
<td>10</td>
<td>[8, 9, 10, 11]</td>
</tr>
<tr>
<td>11</td>
<td>[12, 13, 14, 15]</td>
</tr>
</tbody>
</table>
Practice Questions:

- Suppose a computer’s address size is k bits (using byte addressing), the cache size is S bytes, the block size is B bytes, and the cache is A-way set associative. Assume that B is a power of 2, so \( B = 2^b \). Figure out what the following quantities are in terms of S, B, A, b, and k: the number of sets in the cache, the number of index bits in the address, and the number of bits needed to implement the cache.

\[
\text{Number of sets in the cache:} \quad \text{Sets/cache} = \frac{\text{(Bytes/cache)}}{\text{(Blocks/set)} \times \text{(Bytes/block)}}
\]

\[
= \frac{S}{AB}
\]

Number of address bits needed to index a particular set of the cache:

\[
\text{Cache set index bits} = \log_2 \left( \text{Sets/cache} \right)
\]

\[
= \log_2 \left( \frac{S}{AB} \right)
\]

\[
= \log_2 \left( \frac{S}{A} \right) - b
\]
Practice Questions:

Number of bits needed to implement the cache:
Tag address bits/block = (Total address bits) − (Cache set index bits)
− (Block offset bits)
= \(k - \left(\log_2\left(\frac{S}{A}\right) - b\right) - b\)

= \(k - \log_2\left(\frac{S}{A}\right)\)

Number of bits needed to implement the cache = sets/cache \times associativity \times (data + tag + valid):

= \(\frac{S}{AB} \times A \times \left(8 \times B + k - \log_2\left(\frac{S}{A}\right) + 1\right)\)

= \(\frac{S}{B} \times \left(8B + k - \log_2\left(\frac{S}{A}\right) + 1\right)\)