Lecture 5
Cache Operation

ECE 463/521
Fall 2002
Edward F. Gehringer
Based on notes by Drs. Eric Rotenberg & Tom Conte of NCSU

Outline

• Review of cache parameters
• Example of operation of a direct-mapped cache.
• Example of operation of a set-associative cache.
• Simulating a generic cache.
• Write-through vs. write-back caches.
• Write-allocate vs. no write-allocate.
• Victim caches.

Cache Parameters

• $SIZE$ = total amount of cache data storage, in bytes
• $BLOCKSIZE$ = total number of bytes in a single block
• $ASSOC$ = associativity, i.e., # of blocks in a set
Cache Parameters (cont.)

- Equation for # of cache blocks in cache:
  \[ \frac{\text{SIZE}}{\text{BLOCKSIZE}} \]

- Equation for # of sets in cache:
  \[ \frac{\text{SIZE}}{\text{ASSOC} \times \text{BLOCKSIZE}} \]

Address Fields

- Tag field is compared to the tag(s) of the indexed cache line(s).
  - If it matches, block is there (hit).
  - If it doesn't match, block is not there (miss).

- Used to look up a "set," whose lines contain one or more memory blocks. (The # of blocks per set is the "associativity").

- Once a block is found, the offset selects a particular byte or word of data in the block.

Address Fields (cont.)

- Widths of address fields (# bits)
  \[ \begin{align*}
  \# \text{ index bits} &= \log_2(\# \text{ sets}) \\
  \# \text{ block offset bits} &= \log_2(\text{block size}) \\
  \# \text{ tag bits} &= 32 - \# \text{ index bits} - \# \text{ block offset bits}
  \end{align*} \]

Assuming 32-bit addresses
Example of cache operation

- **Example:** Processor accesses a 256-byte direct-mapped cache, which has 32-byte lines, with the following sequence of addresses.
  - Show contents of the cache after each access.
  - Count # of hits and # of replacements.

Example address sequence

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Tag (hex)</th>
<th>Index &amp; offset</th>
<th>Index</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00001000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00010000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00020000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00030000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00040000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00050000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00060000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example (cont.)

\[ \text{SIZE} = \frac{256}{32} = 8 \]

\[ \text{# sets} = \log_2(\text{SETS}) = \log_2(8) = 3 \]

\[ \text{# block offset bits} = \log_2(\text{block size}) = \log_2(32 \text{ bytes}) = 5 \]

\[ \text{# tag bits} = \text{total # address bits} - \text{# index bits} - \text{# block offset bits} = 32 \text{ bits} - 3 \text{ bits} - 5 \text{ bits} = 24 \]

Thus, the top 6 nibbles (24 bits) of address form the tag and lower 2 nibbles (8 bits) of address form the index and block offset fields.

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Tag (hex)</th>
<th>Index &amp; offset bits (binary)</th>
<th>Index (decimal)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF040F0</td>
<td>0xFFFF040</td>
<td>1110, 0000</td>
<td>7</td>
<td>Miss</td>
</tr>
<tr>
<td>0xFFFF000FC</td>
<td>0xFFFF000</td>
<td>0101, 1100</td>
<td>2</td>
<td>Miss</td>
</tr>
<tr>
<td>0xFFFF040F2</td>
<td>0xFFFF040</td>
<td>1110, 0010</td>
<td>7</td>
<td>Hit</td>
</tr>
<tr>
<td>0xFFFF040F2</td>
<td>0xFFFF040</td>
<td>1110, 1000</td>
<td>7</td>
<td>Hit</td>
</tr>
<tr>
<td>0x00101078</td>
<td>0x001010</td>
<td>0111, 1000</td>
<td>3</td>
<td>Miss</td>
</tr>
<tr>
<td>0x002132E0</td>
<td>0x002132</td>
<td>1110, 0000</td>
<td>3</td>
<td>Miss/Replace</td>
</tr>
<tr>
<td>0x00101064</td>
<td>0x001010</td>
<td>0110, 0100</td>
<td>3</td>
<td>Hit</td>
</tr>
<tr>
<td>0x0012355C</td>
<td>0x001235</td>
<td>0101, 1100</td>
<td>3</td>
<td>Miss/Replace</td>
</tr>
<tr>
<td>0x00122544</td>
<td>0x001225</td>
<td>0110, 0100</td>
<td>2</td>
<td>Hit</td>
</tr>
</tbody>
</table>

Get block from memory (slow)
Get block from memory (slow)
Get block from memory (slow)

Get block from memory (slow)

Get block from memory (slow)

HIT
Set-Associative Example

- **Example:** Processor accesses a 256-byte 2-way set-associative cache, which has block size of 32 bytes, with the following sequence of addresses.
  - Show contents of cache after each access.
  - Count # of hits and # of replacements.
Example (cont.)

\[
\begin{array}{c|c|c|c}
\hline
\text{Address (hex)} & \text{Tag (hex)} & \text{Index \\ \\
& \text{offset (binary)} & \text{offset (decimal)} \\
\hline
\text{0x000000} & \text{0x1F0000} & 0000000 & 0 & \text{Miss} \\
\text{0x002000} & \text{0x170200} & 00010000 & 2 & \text{Miss} \\
\text{0x001000} & \text{0x002000} & 00000000 & 0 & \text{Miss} \\
\text{0x000000} & \text{0x1F0081} & 00000000 & 0 & \text{Hit} \\
\text{0x000000} & \text{0x000020} & 00000000 & 0 & \text{Hit} \\
\text{0x000000} & \text{0x000030} & 00000000 & 0 & \text{Miss/Replace} \\
\text{0x000000} & \text{0x000000} & 00000000 & 0 & \text{Hit} \\
\hline
\end{array}
\]

Notes:
- \( \# \text{sets} = \log_2(\# \text{sets}) = \log_2(4) = 2 \)
- \( \# \text{block offset bits} = \log_2(\text{block size}) = \log_2(32 \text{ bytes}) = 5 \)
- \( \# \text{tag bits} = \text{total \# address bits} - \# \text{index bits} - \# \text{block offset bits} = 32 \text{ bits} - 2 \text{ bits} - 5 \text{ bits} = 25 \)
Generic Cache

- Every cache is an $n$-way set-associative cache.
  1. Direct-mapped:
     - 1-way set-associative
  2. $n$-way set-associative:
     - $n$-way set-associative
  3. Fully-associative:
     - $n$-way set-associative, where there is only 1 set containing $n$ blocks
     - # index bits = $\log_2(n) = 0$ (equation still works!)

Generic Cache (cont.)

- The same equations hold for any cache type
- Equation for # of cache blocks in cache:
  $\frac{\text{SIZE}}{\text{BLOCKSIZE}}$
- Equation for # of sets in cache:
  $\frac{\text{SIZE}}{\text{ASSOC} \times \text{BLOCKSIZE} \times \text{ASSOC}}$
- Fully-associative: $\text{ASSOC} = \# \text{ cache blocks}$
Generic Cache (cont.)

- What this means
  - You don’t have to treat the three cache types differently in your simulator!
  - Support a generic $n$-way set-associative cache
  - You don’t have to specifically worry about the two extremes (direct-mapped & fully-associative).
  - E.g., even DM cache has LRU information (in simulator), even though it is not used.

Replacement Policy

- Which block in a set should be replaced when a new block has to be allocated?
  - LRU (least recently used) is the most common policy.
  - Implementation:
    - Real hardware maintains LRU bits with each block in set.
    - Simulator:
      - Cheat using “sequence numbers” as timestamps.
      - Makes simulation easier – gives same effect as hardware.

LRU Example

- Assume that blocks A, B, C, D, and E all map to the same set
- Trace: A B C D D B D E
- Assign sequence numbers (increment by 1 for each new access). This yields ...

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(0)</th>
<th>D(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1)</td>
<td>B(1)</td>
</tr>
<tr>
<td>C(2)</td>
<td>B(2)</td>
</tr>
<tr>
<td>D(3)</td>
<td>D(3)</td>
</tr>
</tbody>
</table>
Handling Writes

- What happens when a write occurs?
- Two issues
  1. Is just the cache updated with new data, or are lower levels of memory hierarchy updated at same time?
  2. Do we allocate a new block in the cache if the write misses?

The Write-Update Question

- Write-through (WT) policy: Writes that go to the cache are also "written through" to the next level in the memory hierarchy.

  ![Diagram of write-through policy]

The Write-Update Question, cont.

- Write-back (WB) policy: Writes go only to the cache, and are not (immediately) written through to the next level of the hierarchy.

  ![Diagram of write-back policy]
The Write-Update Question, cont.

- Write-back, cont.
  - What happens when a line previously written to needs to be replaced?
  1. We need to have a “dirty bit” (D) with each line in the cache, and set it when line is written to.
  2. When a dirty block is replaced, we need to write the entire block back to next level of memory ("write-back").

The Write-Update Question, cont.

- With the write-back policy, replacement of a dirty block triggers a writeback of the entire line.

The Write-Allocation Question

- Write-Allocate (WA)
  - Bring the block into the cache if the write misses (handled just like a read miss).
  - Typically used with write-back policy: WBWA
- Write-No-Allocate (NA)
  - Do not bring the block into the cache if the write misses.
  - Must be used in conjunction with write-through: WTNA.
The Write-Allocation Question, cont.

• WTNA (scenario: the write misses)
  - cache
  - write miss
  - next level in memory hierarchy

The Write Allocation Question, cont.

• WBWA (scenario: the write misses)
  - cache
  - next level in memory hierarchy

Victim Caches

• A victim cache is a small fully-associative cache that sits alongside primary cache
  - E.g., holds 2–16 cache blocks
  - Management is slightly "weird" compared to conventional caches
    - When main cache evicts (replaces) a block, the victim cache will take the evicted (replaced) block.
    - The evicted block is called the "victim."
    - When the main cache misses, the victim cache is searched for recently discarded blocks. A victim cache hit means the main cache doesn't have to go to memory for the block.
Victim-Cache Example

- Suppose we have a 2-entry victim cache
  - It initially holds blocks X and Y.
  - Y is the LRU block in the victim cache.
- The main cache is direct-mapped.
  - Blocks A and B map to the same set in main cache
  - Trace: A B A B A B ...

Victim-Cache Example, cont.

1. B misses in main cache and evicts A.
2. A goes to victim cache & replaces Y. (the previous LRU)
3. X becomes LRU.
Victim-Cache Example, cont.

1. A misses in L1 but hits in victim cache, so A and B swap positions:
2. A is moved from victim cache to L1, and B (the victim) goes to victim cache where A was located. (Note: We don’t replace the LRU block, X, in case of victim cache hit)

Victim Cache – Why?

- Direct-mapped caches suffer badly from repeated conflicts
  - Victim cache provides illusion of set associativity.
  - A poor-man’s version of set associativity.
  - A victim cache does not have to be large to be effective; even a 4–8 entry victim cache will frequently remove > 50% of conflict misses.