Caches III
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton
Michael Zhang
Parker DeWilde
Ryan Wong
Sam Gehman
Sam Wolfson
Savanna Yee
Vinny Palaniappan
Administrivia

- Midterm regrade requests due end of tonight
- Lab 3 due Friday
- HW 4 is released, due next Friday (11/17)
- No lecture on Friday – Veteran’s Day!
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- Program optimizations that consider caches
Associativity

- What if we could store data in any place in the cache?
  - More complicated hardware = more power consumed, slower
- So we combine the two ideas:
  - Each address maps to exactly one set
  - Each set can store block in more than one way
Cache Organization (3)

- **Associativity** ($E$): # of ways for each set
  - Such a cache is called an “$E$-way set associative cache”
  - We now index into cache sets, of which there are $C/K/E$
  - Use lowest $\log_2(C/K/E) = s$ bits of block address
    - Direct-mapped: $E = 1$, so $s = \log_2(C/K)$ as we saw previously
    - Fully associative: $E = C/K$, so $s = 0$ bits

---

### Diagram

- **Tag ($t$)**: Used for tag comparison
- **Index ($s$)**: Selects the set
- **Offset ($k$)**: Selects the byte from block

**Increasing associativity**

- **Fully associative**: (only one set)
- **Increasing associativity**

**Decreasing associativity**

- **Direct mapped**: (only one way)
Example Placement

- Where would data from address \(0x1833\) be placed?
  - Binary: \(0b\ 0001\ 1000\ 0011\ 0011\)

\[
t = m - s - k \quad s = \log_2(C/K/E) \quad k = \log_2(K)
\]

- \(m\)-bit address:
  - Tag \((t)\)
  - Index \((s)\)
  - Offset \((k)\)

\(s = ?\)
- Direct-mapped

\(s = ?\)
- 2-way set associative

\(s = ?\)
- 4-way set associative
Block Replacement

- *Any* empty block in the correct set may be used to store block.
- If there are no empty blocks, which one should we replace?
  - No choice for direct-mapped caches.
  - Caches typically use something close to *least recently used (LRU)* (hardware usually implements “*not most recently used*”).

<table>
<thead>
<tr>
<th>Direct-mapped</th>
<th>2-way set associative</th>
<th>4-way set associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Tag</td>
<td>Data</td>
</tr>
<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>
Peer Instruction Question

- We have a cache of size 2 KiB with block size of 128 B. If our cache has 2 sets, what is its associativity?
  A. 2
  B. 4
  C. 8
  D. 16
  E. We’re lost...

- If addresses are 16 bits wide, how wide is the Tag field?
General Cache Organization \((S, E, K)\)

- \(E\) = blocks/lines per set
- \(S\) = \# sets = \(2^s\)
- \(K\) = bytes per block
- Cache size:
  \[ C = K \times E \times S \] data bytes (doesn’t include V or Tag)
Notation Review

- We just introduced a lot of new variable names!
  - Please be mindful of block size notation when you look at past exam questions or are watching videos.

<table>
<thead>
<tr>
<th>Variable</th>
<th>This Quarter</th>
<th>Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>$K (B$ in book)</td>
<td>$M = 2^m \leftrightarrow m = \log_2 M$</td>
</tr>
<tr>
<td>Cache size</td>
<td>$C$</td>
<td>$S = 2^s \leftrightarrow s = \log_2 S$</td>
</tr>
<tr>
<td>Associativity</td>
<td>$E$</td>
<td>$K = 2^K \leftrightarrow K = \log_2 K$</td>
</tr>
<tr>
<td>Number of Sets</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>Address space</td>
<td>$M$</td>
<td>$C = K \times E \times S$</td>
</tr>
<tr>
<td>Address width</td>
<td>$m$</td>
<td>$s = \log_2 (C/K/E)$</td>
</tr>
<tr>
<td>Tag field width</td>
<td>$t$</td>
<td>$m = t + s + k$</td>
</tr>
<tr>
<td>Index field width</td>
<td>$s$</td>
<td></td>
</tr>
<tr>
<td>Offset field width</td>
<td>$k$ ($b$ in book)</td>
<td></td>
</tr>
</tbody>
</table>
Cache Read

1) Locate set
2) Check if any line in set is valid and has matching tag: hit
3) Locate data starting at offset

\[ E = \text{blocks/lines per set} \]

\[ S = \# \text{sets} = 2^s \]

Address of byte in memory:

\[ \text{tag} \quad \text{set index} \quad \text{block offset} \]

data begins at this offset

valid bit

\[ K = \text{bytes per block} \]
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set  
Block Size $K = 8$ B

$S = 2^{s}$ sets

Address of `int`:  

$\text{find set}$
Example: Direct-Mapped Cache \((E = 1)\)

Direct-mapped: One line per set
Block Size \(K = 8\) B

Address of \texttt{int}:

\begin{itemize}
  \item \texttt{valid?} + \texttt{match?: yes = hit}
  \item Block offset
  \item Tag bits: 0...01 100
\end{itemize}
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B

No match? Then old line gets evicted and replaced

This is why we want alignment!
Example: Set-Associative Cache \((E = 2)\)

2-way: Two lines per set
Block Size \(K = 8\) B

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v</td>
<td>tag</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Address of short int:

```
| bits | 0...01 | 100 |
```

find set
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

valid? + match: yes = hit

Address of `short int`:

```
short int: 0...01 100
```

Comparison bits:

```
0 1 2 3 4 5 6 7
```

Block offset:

```
0 1 2 3 4 5 6 7
```

v tag 0 1 2 3 4 5 6 7

compare both
Example: Set-Associative Cache ($E = 2$)

2-way: Two lines per set
Block Size $K = 8$ B

No match?
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Address of `short int`:

No match?
Types of Cache Misses: 3 C’s!

- **Compulsory** (cold) miss
  - Occurs on first access to a block

- **Conflict** miss
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - *e.g.* referencing blocks 0, 8, 0, 8, ... could miss every time
  - Direct-mapped caches have more conflict misses than $E$-way set-associative (where $E > 1$)

- **Capacity** miss
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won’t fit, even if cache was *fully-associative*)
  - **Note:** *Fully-associative* only has Compulsory and Capacity misses
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What to do on a write-hit?
  - Write-through: write immediately to next level
  - Write-back: defer write to next level until line is evicted (replaced)
    - Must track which cache lines have been modified ("dirty bit")
- What to do on a write-miss?
  - Write-allocate: ("fetch on write") load into cache, update line in cache
    - Good if more writes or reads to the location follow
  - No-write-allocate: ("write around") just write immediately to memory
- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Write-back, write-allocate example

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.
Write-back, write-allocate example

mov 0xFACE, F

Cache

Memory
Write-back, write-allocate example

```c
mov 0xFACE, F
```

Step 1: Bring F into cache
Write-back, write-allocate example

```plaintext
mov 0xFACE, F
```

**Step 2: Write 0xFACE to cache only and set dirty bit**
Write-back, write-allocate example

\[ \text{mov 0xFACE, F} \quad \text{mov 0xFEED, F} \]

Cache

Memory

Write hit!
Write 0xFEED to cache only

dirty bit
Write-back, write-allocate example

\textbf{mov} 0xFACE, F \quad \textbf{mov} 0xFEED, F \quad \textbf{mov} G, \%rax

\begin{itemize}
  \item \textbf{mov} 0xFEED, F
  \item \textbf{mov} 0xCafe, F
  \item \textbf{mov} G, \%rax
\end{itemize}

Dirty bit

Cache

Memory
Write-back, write-allocate example

mov 0xFACE, F  
mov 0xFEED, F  
mov G, %rax

1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into %rax
Peer Instruction Question

- Which of the following cache statements is FALSE?

A. We can reduce compulsory misses by decreasing our block size

B. We can reduce conflict misses by increasing associativity

C. A write-back cache will save time for code with good temporal locality on writes

D. A write-through cache will always match data with the memory hierarchy level below it

E. We’re lost...
Example Cache Parameters Problem

- 1 MiB address space, 125 cycles to go to memory.

Fill in the following table:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>4 KiB</td>
</tr>
<tr>
<td>Block Size</td>
<td>16 B</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>Hit Time</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>20%</td>
</tr>
<tr>
<td>Write Policy</td>
<td>Write-through</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Tag Bits</td>
<td></td>
</tr>
<tr>
<td>Index Bits</td>
<td></td>
</tr>
<tr>
<td>Offset Bits</td>
<td></td>
</tr>
<tr>
<td>AMAT</td>
<td></td>
</tr>
</tbody>
</table>
Example Code Analysis Problem

- Assuming the cache starts **cold** (all blocks invalid), calculate the **miss rate** for the following loop:
  - \( m = 20 \text{ bits}, C = 4 \text{ KiB}, K = 16 \text{ B}, E = 4 \)

```c
#define AR_SIZE 2048
int int_ar[AR_SIZE], sum=0; // &int_ar=0x80000
for (int i=0; i<AR_SIZE; i++)
    sum += int_ar[i];
for (int j=AR_SIZE-1; j>=0; j--)
    sum += int_ar[i];
```