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

https://what-if.xkcd.com/111/
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

<table>
<thead>
<tr>
<th></th>
<th>1-way:</th>
<th>2-way:</th>
<th>4-way:</th>
<th>8-way:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
<td>1 set, 8 blocks</td>
</tr>
</tbody>
</table>

- 1-way: direct mapped
- 2-way: solves conflict problem!
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 $C/K/E=S$ sets
  - 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

- Decreasing associativity
- Direct mapped (only one way)
- Increasing associativity
- Fully associative (only one set)

**Note:** The textbook uses “b” for offset bits
Example Placement

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

  \[ t = m - s - k \]
  \[ s = \log_2(C/K/E) \]
  \[ k = \log_2(K) = 4 \]

  \( m \)-bit address:
  \[
  \begin{array}{c|c|c}
  \text{Tag (t)} & \text{Index (s)} & \text{Offset (k)} \\
  \end{array}
  \]

  \( s = ? \log_2(8/1) = 3 \) bit
  - Direct-mapped (E=1)

  \( s = ? \log_2(8/2) = 2 \) bit
  - 2-way set associative (E=2)

  \( s = ? \log_2(8/4) = 1 \) bit
  - 4-way set associative (E=4)

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6</td>
<td>✓</td>
<td></td>
<td>6</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>7</td>
<td>✓</td>
<td></td>
<td>7</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
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>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>
<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>

Direct-mapped

<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>

2-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>

4-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>
</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?
  - Vote at http://PollEv.com/justinh
  - A. 2
  - B. 4
  - C. 8
  - D. 16
  - E. We’re lost...

- If addresses are 16 bits wide, how wide is the Tag field?
  \[ k = \log_2(K) = 7 \text{ bits}, \quad s = \log_2(S) = 1 \text{ bit}, \quad t = m - s - k = 8 \text{ bits} \]
General Cache Organization $(S, E, K)$

- $E$ = blocks/lines per set
- $S$ = # sets = $2^s$
- $K$ = bytes per block
- $V$ = valid bit
- $C = K \times E \times S$ data bytes (doesn’t include $V$ or Tag)

Cache size:

- Set (contains E lines)
- Line (block plus management bits)
- Layers of a cache:
  - Block (data)
  - Line
  - Set
  - Cache
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>$C = K \times E \times S$</td>
</tr>
<tr>
<td>Address space</td>
<td>$M$</td>
<td>$s = \log_2 (C/K/E)$</td>
</tr>
<tr>
<td>Address width</td>
<td>$m$</td>
<td>$m = t + s + k$</td>
</tr>
<tr>
<td>Tag field width</td>
<td>$t$</td>
<td></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

Address of byte in memory:

$$S = \# \text{ sets} = 2^s$$

$$E = \text{blocks/lines per set}$$

$$K = \text{bytes per block}$$

data begins at this offset
Example: Direct-Mapped Cache ($E = 1$)

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

$S = 2^s$ sets

Address of \texttt{int}:

\begin{tabular}{c}
  \texttt{bits}  \\
  0...01  \\
  100
\end{tabular}

find set
Example: Direct-Mapped Cache ($E = 1$)

Direct-mapped: One line per set
Block Size $K = 8$ B
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
Example: Set-Associative Cache ($E = 2$)

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

Address of short int:

| bits | 0...01 | 100 |

Set 0
Set 1
Set 2

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

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

Address of `short int`:

```plaintext
0...01 100
```

valid? + match: yes = hit

compare both

Block offset
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 \texttt{short int}:

\texttt{bits} 0...01 100

\texttt{short int} \((2\) B) is here
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? (block/data already in $)$
  - 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? (block/data not currently in $)$
  - 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
```

1. Check cache for F → miss
2. Pull block into $, then write

```
G 0xBEEF
```

```
Memory
```

```
F 0xCAFE
G 0xBEEF
```

Dirty bit

The same, so
Write-back, write-allocate example

mov 0xFACE, F

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

```plaintext
mov 0xFACE, F
```
Write-back, write-allocate example

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

Write hit!
Write 0xFEED to cache only

Dirty bit
Write-back, write-allocate example

```
mov 0xFACE, F  mov 0xFEED, F
```

Memory

```
F
0xCAFE

G
0xBEEF
```

Cache

```
F
0xFEED

1
dirty bit
```

mov G, %rax

read miss
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></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size (C)</td>
<td>4 KiB = $2^{12}$ B</td>
</tr>
<tr>
<td>Block Size (K)</td>
<td>16 B = $2^{4}$ B</td>
</tr>
<tr>
<td>Associativity (E)</td>
<td>4-way = $2^{2}$</td>
</tr>
<tr>
<td>Hit Time (HT)</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Miss Rate (MR)</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>10</td>
</tr>
<tr>
<td>Index Bits</td>
<td>6</td>
</tr>
<tr>
<td>Offset Bits</td>
<td>4</td>
</tr>
<tr>
<td>AMAT</td>
<td>$3 + 0.2 \times 125 = 28$</td>
</tr>
</tbody>
</table>
Example Code Analysis Problem

- Assuming the cache starts **cold** (all blocks invalid), calculate the **miss rate** for the following loop:
  - \( m = 20 \) bits, \( C = 4 \) KiB, \( K = 16 \) B, \( E = 4 \)
  - \#define AR_SIZE 2048 = \( 2^\text{13} \) B of data

```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]; // read i
for (int j=AR_SIZE-1; j>=0; j--)
    sum += int_ar[i]; // read i
```

- Miss rate: \( \frac{3}{4} \) (half of \( \text{int}_\text{ar}[] \))
- \( \ell = 10, s = 6, k = 4 \)
- \( \text{int}_\text{ar}[0] \) accesses first 4 B (offset 0) of a cache block in set 0.

- Loop 1: never re-visit blocks. First half of loop fills entire symmetric 4 KiB with data from lower half of \( \text{int}_\text{ar}[] \).
  - Second half of loop replaces entire 4 KiB data with upper half of \( \text{int}_\text{ar}[] \).

- Loop 2: first half of loop uses upper half of \( \text{int}_\text{ar}[] \), which is already in the cache (miss rate of 0). Second half of loop replaces entire 4 KiB data with lower half of \( \text{int}_\text{ar}[] \).