Direct-Mapped and Set Associative Caches

Instructor: Steven Ho
Great Idea #3: Principle of Locality/Memory Hierarchy

- Processor Register
- CPU Cache (L1, L2, L3)
- Physical Memory (RAM)
- Solid State Memory (SSD, Flash Drive)
- Virtual Memory (File-Based Memory)
- Mechanical Hard Drives

- Super Fast
- Super Expensive
- Tiny Capacity
- Faster
- Expensive
- Small Capacity
- Fast
- Priced Reasonably
- Average Capacity
- Average Speed
- Priced Reasonably
- Average Capacity
- Slow
- Cheap
- Large Capacity
Extended Review of Last Lecture

• Why have caches?
  – Intermediate level between CPU and memory
  – In-between in size, cost, and speed
• Memory (hierarchy, organization, structures) set up to exploit temporal and spatial locality
  – Temporal: If accessed, will access again soon
  – Spatial: If accessed, will access others around it
• Caches hold a subset of memory (in blocks)
  – We are studying how they are designed for fast and efficient operation (lookup, access, storage)
Extended Review of Last Lecture

- Fully Associative Caches:
  - Every block can go in any slot
    - Use random or LRU replacement policy when cache full
  - Memory address breakdown (on request)
    - Tag field is unique identifier (which block is currently in slot)
    - Offset field indexes into block (by bytes)
  - Each cache slot holds block data, tag, valid bit, and dirty bit (dirty bit is only for write-back)
    - The whole cache maintains LRU bits
Extended Review of Last Lecture

• On memory access (read or write):
  1) Look at ALL cache slots in parallel
  2) If Valid bit is 0, then ignore (garbage)
  3) If Valid bit is 1 and Tag matches, then use that data

• On write, set Dirty bit if write-back
Extended Review of Last Lecture

- Fully associative cache layout in our example
  - 6-bit address space, 16-byte cache with 4-byte blocks
  - How many blocks do we have? \( C/K = 4 \) blocks
  - LRU replacement (2 bits)
  - Offset – 2 bits, Tag – 4 bits

\[
\begin{array}{c|c|cccc|c}
\text{Slot} & \text{V} & \text{Tag} & 00 & 01 & 10 & 11 & \text{LRU} \\
0 & X & XXXX & 0x?? & 0x?? & 0x?? & 0x?? & XX \\
1 & X & XXXX & 0x?? & 0x?? & 0x?? & 0x?? & XX \\
2 & X & XXXX & 0x?? & 0x?? & 0x?? & 0x?? & XX \\
3 & X & XXXX & 0x?? & 0x?? & 0x?? & 0x?? & XX \\
\end{array}
\]

Yesterday’s example was write through and looked like this
1) Consider the following *addresses* being requested:

Starting with a cold cache:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

### 000000

**0 miss**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

### 000010

**2 hit**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

### 000000

**0 hit**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

### 000010

**2 hit**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

### 000000

**0 hit**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>
FA Cache Examples (3/4)

1) Consider the following *addresses* being requested:

Starting with a cold cache:

```
   0  2  2  0  16  20  8  4
M  H  H  H
```

- **010000**
  - 16 miss
  - Starting state:
    - 0 0000 0x?? 0x?? 0x?? 0x??
    - 0 0000 0x?? 0x?? 0x?? 0x??

- **010100**
  - 20 miss
  - Starting state:
    - 0 0000 0x?? 0x?? 0x?? 0x??

- **001000**
  - 8 miss
  - Starting state:

- **000100**
  - 4 miss
  - Starting state:

- 8 requests, 5 misses – ordering matters!
3) Original sequence, but double block size to 8B

Starting with a cold cache:

0 2 4 8 20 16 0 2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hit</td>
<td>2</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hit</td>
<td>4</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

|--------|---|------|------|------|------|------|------|------|------|
3) Original sequence, but double block size

Starting with a cold cache:

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>20</th>
<th>16</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Misses:
  - 010100
  - 010000
  - 000000
  - 000010

- Hits:
  - 010100
  - 010000
  - 000000
  - 000010

- 8 requests, 4 misses – cache parameters matter!
Question:
Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

(A) 0 2 12 4 16 8 0 6
(B) 0 8 4 16 0 12 6 2
(C) 6 12 4 8 2 16 0 0
(D) 2 8 0 4 6 16 12 0
Question:
Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

(A) 0 2 12 4 16 8 0 6
(B) 0 8 4 16 0 12 6 2
(C) 6 12 4 8 2 16 0 0
(D) 2 8 0 4 6 16 12 0

Correct answer: (D)
Question:
Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

**LRU**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) 0 2 12 4 16 8 0 6
(B) 0 8 4 16 0 12 6 2
(C) 6 12 4 8 2 16 0 0
(D) 2 8 0 4 6 16 12 0
Memory Accesses

• The picture so far:
Handling Write Hits

• Write hits (D$)

1) Write-Through Policy: Always write data to cache and to memory (through cache)
   • Forces cache and memory to always be consistent
   • Slow! (every memory access is long)
   • Include a Write Buffer that updates memory in parallel with processor

Assume present in all schemes when writing to memory
Handling Write Hits

• Write hits (D$)

2) Write-Back Policy: Write data **only to cache**, then update memory when block is removed
   • Allows cache and memory to be inconsistent
   • Multiple writes collected in cache; single write to memory per block
   • **Dirty bit:** Extra bit per cache row that is set if block was written to (is “dirty”) and needs to be written back
Handling Cache Misses

• Miss penalty grows as block size does
• Read misses (I$ and D$)
  – Stall execution, fetch block from memory, put in cache, send requested data to processor, resume
• Write misses (D$)
  – Always have to update block from memory
  – We have to make a choice:
    • Carry the updated block into cache or not?
Write Allocate

• **Write Allocate** policy: when we bring the block into the cache after a write miss

• **No Write Allocate** policy: only change main memory after a write miss
  – Write allocate almost always paired with write-back
    • Eg: Accessing same address many times -> cache it
  – No write allocate typically paired with write-through
    • Eg: Infrequent/random writes -> don’t bother caching it
Updated Cache Picture

- Fully associative, write through
  - Same as our simplified examples from before
- Fully associative, write back

<table>
<thead>
<tr>
<th>Slot</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>XXXX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>XXXX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td>XXXX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>XXXX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

- Write miss procedure (write allocate or not) only affects behavior, not design
How do we use this thing?

• Nothing changes from the programmer’s perspective
  – Still just issuing \texttt{lw} and \texttt{sw} instructions

• The rest is handled in hardware:
  – Checking the cache
  – Extracting the data using the offset

• Why should a programmer care?
  – Understanding cache parameters = faster programs
Agenda

• Review of yesterday
• Administrativia
• Direct-Mapped Caches
• Set Associative Caches
• Cache Performance
Administrivia

• HW3/4 Due today
• HW5 Released, due next Monday (7/23)
• Project 3 Due Friday (7/20)
  – Parties tonight @Soda 405/411 and Friday @Woz (4-6pm for both)
  – If you ask for help please diagnose problem spots
• Guerilla Session on Wed. 4-6pm @Soda 405
• Midterm 2 is coming up! Next Wed. in lecture
  – Covering up to Performance
  – Review Session Sunday 2-4pm @GPB 100
Direct-Mapped Caches (1/3)

• Each memory block is mapped to exactly one slot in the cache (*direct-mapped*)
  – Every block has only one “home”
  – Use hash function to determine which slot

• Comparison with fully associative
  – Check just one slot for a block (faster!)
  – No replacement policy necessary
  – Access pattern may leave empty slots in cache
Direct-Mapped Caches (2/3)

• **Offset field** remains the same as before

• **Recall:** blocks consist of adjacent bytes
  – Do we want adjacent blocks to map to same slot?
  – **Index field:** Apply hash function to block address to determine *which slot* the block goes in
    • \((\text{block address}) \mod (\# \text{ of blocks in the cache})\)

• **Tag field** maintains same function (identifier), but is now shorter
TIO Address Breakdown

• Memory address fields:

<table>
<thead>
<tr>
<th></th>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- $T$ bits
- $I$ bits
- $O$ bits

• Meaning of the field sizes:
  - $O$ bits $\leftrightarrow 2^O$ bytes/block $= 2^{O-2}$ words/block
  - $I$ bits $\leftrightarrow 2^I$ slots in cache $= \text{cache size} / \text{block size}$
  - $T$ bits $= A - I - O$, where $A = \# \text{ of address bits}$
    (A = 32 here)
Direct-Mapped Caches (3/3)

• What’s actually in the cache?
  – Block of data \((8 \times K = 8 \times 2^O)\) bits
  – Tag field of address as identifier \((T)\) bits
  – Valid bit (1 bit)
  – Dirty bit (1 bit if write-back)
  – No replacement management bits!

• Total bits in cache = \# slots \times (8\times K + T + 1 + 1) 
  = 2^l \times (8\times 2^O + T + 1 + 1) \text{ bits}
DM Cache Example (1/5)

• Cache parameters:
  – Direct-mapped, address space of 64B, block size of 4B, cache size of 16B, write-through

• TIO Breakdown:
  – \( O = \log_2(4) = 2 \)
  – Cache size / block size = 16/4 = 4, so \( I = \log_2(4) = 2 \)
  – \( A = \log_2(64) = 6 \) bits, so \( T = 6 - 2 - 2 = 2 \)

• Bits in cache = \( 2^2 \times (8 \times 2^2 + 2 + 1) = 140 \) bits

Memory Addresses: \( XX|XX|XX \)

Block address
DM Cache Example (2/5)

• Cache parameters:
  – Direct-mapped, address space of 64B, block size of 4B, cache size of 16B, write-through
  – Offset – 2 bits, Index – 2 bits, Tag – 2 bits

<table>
<thead>
<tr>
<th>Index</th>
<th>Offset</th>
<th>V</th>
<th>Tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>X</td>
<td>XX</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

• 35 bits per index(slot), 140 bits to implement
Main Memory:

Which blocks map to each row of the cache? (see colors)

On a memory request: (let’s say 001011 two)

1) Take Index field (10)
2) Check if Valid bit is true in that row of cache
3) If valid, then check if Tag matches

Cache slots exactly match the Index field

Main Memory shown in blocks, so offset bits not shown (x’s)
DM Cache Example (4/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:

0 2 4 8 20 16 0 2

000000
0 miss

01 0 00 0x?? 0x?? 0x?? 0x??
10 0 00 0x?? 0x?? 0x?? 0x??
11 0 00 0x?? 0x?? 0x?? 0x??

000100
4 miss

10 0 00 0x?? 0x?? 0x?? 0x??
11 0 00 0x?? 0x?? 0x?? 0x??

000010
2 hit

01 0 00 0x?? 0x?? 0x?? 0x??
10 0 00 0x?? 0x?? 0x?? 0x??
11 0 00 0x?? 0x?? 0x?? 0x??

001000
8 miss

11 0 00 0x?? 0x?? 0x?? 0x??
DM Cache Example (5/5)

- Consider the sequence of memory address accesses
  
  Starting with a cold cache:

  0 2 4 8 20 16 0 2

  010100
  20 miss

  11 0 0 0 0

  0000000
  0 miss

  11 0 0 0 0

  000010
  2 hit

  11 0 0 0 0

- 8 requests, 6 misses – last slot was never used!
Worst-Case for Direct-Mapped

• Cold DM $ that holds four 1-word blocks
• Consider the memory accesses: 0, 16, 0, 16,...

<table>
<thead>
<tr>
<th>Memory Access</th>
<th>Cache State</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>Miss</td>
</tr>
<tr>
<td>00</td>
<td>M[0-3]</td>
</tr>
<tr>
<td>010000</td>
<td>Miss</td>
</tr>
<tr>
<td>00</td>
<td>M[0-3]</td>
</tr>
<tr>
<td>000000</td>
<td>Miss</td>
</tr>
<tr>
<td>01</td>
<td>M[16-19]</td>
</tr>
</tbody>
</table>

...
Comparison So Far

• Fully associative
  – Block can go into *any* slot
  – Must check ALL cache slots on request ("slow")
  – TO breakdown (i.e. I = 0 bits)
  – “Worst case” still fills cache (more efficient)

• Direct-mapped
  – Block goes into *one specific* slot (set by Index field)
  – Only check ONE cache slot on request ("fast")
  – TIO breakdown
  – “Worst case” may only use 1 slot (less efficient)
## Meet the Staff

<table>
<thead>
<tr>
<th></th>
<th>Sukrit</th>
<th>Suvansh</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Favorite Villain</strong></td>
<td>The Lannisters</td>
<td>Logisim [De]Evolution</td>
</tr>
<tr>
<td><strong>What would you protest</strong></td>
<td>Prerequisite enforcement</td>
<td>CS Design requirement</td>
</tr>
<tr>
<td><strong>What are you passionate about?</strong></td>
<td>Musiccc</td>
<td>American football</td>
</tr>
<tr>
<td><strong>What you'd want to be famous for?</strong></td>
<td>Arora's Algorithm</td>
<td>Facial Hair</td>
</tr>
</tbody>
</table>
Agenda

• Review of yesterday
• Administrivia
• Direct-Mapped Caches
• Set Associative Caches
• Cache Performance
Set Associative Caches

• Compromise!
  – More flexible than DM, more structured than FA

• *N-way set-associative*: Divide $ into sets, each of which consists of N slots
  – Memory block maps to a set determined by Index field and is placed in any of the N slots of that set
  – Call N the *associativity*
  – New hash function:
    (block address) modulo (# sets in the cache)
  – Replacement policy applies to every set
Effect of Associativity on TIO (1/2)

• Here we assume a cache of fixed size (C)
• **Offset:** # of bytes in a block (same as before)
• **Index:** Instead of pointing to a *slot*, now points to a *set*, so \( I = \log_2(C/\div K/\div N) \)
  – Fully associative (1 set): 0 Index bits!
  – Direct-mapped (N = 1): max Index bits
  – Set associative: somewhere in-between
• **Tag:** Remaining identifier bits \((T = A - I - O)\)
Effect of Associativity on TIO (2/2)

- 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 slots) and halves the number of sets – decreasing the size of the Index by 1 bit and increasing the size of the Tag by 1 bit.

Decreasing associativity

- Direct mapped (only one way)

Increasing associativity

- Fully associative (only one set)
Example: Eight-Block Cache Configs

- Total size of $ = \# sets \times$ associativity
- For fixed $ size, associativity ↑ means $ sets ↓$ and slots per set ↑
- With 8 blocks, an 8-way set associative $ is same as a fully associative $
Block Placement Schemes

- Place memory block 12 in a cache that holds 8 blocks

  **Direct mapped**
  - Block # 0 1 2 3 4 5 6 7
  - Tag 1 2
  - Data
  - Search

  **Set associative**
  - Set # 0 1 2 3
  - Tag 1 2
  - Data
  - Search

  **Fully associative**
  - Data
  - Search

- **Fully associative**: Can go in *any* of the slots (all 1 set)
- **Direct-mapped**: Can only go in slot \((12 \mod 8) = 4\)
- **2-way set associative**: Can go in either slot of set \((12 \mod 4) = 0\)
SA Cache Example (1/5)

• Cache parameters:
  – 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through

• How many sets?
  – $C \div K \div N = 4 \div 1 \div 2 = 2$ sets

• TIO Breakdown:
  – $O = \log_2(4) = 2$, $I = \log_2(2) = 1$, $T = 6 - 1 - 2 = 3$

Memory Addresses: XXX[X][XX]

Block address
SA Cache Example (2/5)

- Cache parameters:
  - 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through
  - Offset – 2 bits, Index – 1 bit, Tag – 3 bits
  - 37 bits per slot, 37*2 = 74 bits per set, 2*74 = 148 bits to implement
SA Cache Example (3/5)

Each block maps into one set (either slot) (see colors)

On a memory request: (let's say 001011<sub>two</sub>)

1) Take Index field (0)
2) For EACH slot in set, check valid bit, then compare Tag

Cache:

<table>
<thead>
<tr>
<th>Set</th>
<th>Slot</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Main Memory shown in blocks, so offset bits not shown (x’s)

Main Memory:

```
0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx
```
SA Cache Example (4/5)

- Consider the sequence of memory address accesses:

Starting with a cold cache:

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>20</th>
<th>16</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0 miss

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

0 miss

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

4 miss

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

0 miss

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

8 miss

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

2 hit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
</tbody>
</table>

8 miss
SA Cache Example (5/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:

0  
     2     4     8     20    16     0     2

**Misses and Hits**

010100  
20 miss

000000  
0  miss

000010  
2  hit

8 requests, 6 misses
Worst Case for Set Associative

- Worst case for DM was repeating pattern of 2 into same cache slot (HR = 0/n)
  - Set associative for N > 1: HR = (n-2)/n
- Worst case for N-way SA with LRU?
  - Repeating pattern of at least N+1 that maps into same set
  - Back to HR = 0:

```
0, 8, 16, 0, 8,
M M M M
... M
```

```
000 M[8-11]
000 M[8-11]
000 M[8-11]
```

Question: What is the TIO breakdown for the following cache?

- 32-bit address space
- 32 KiB 4-way set associative cache
- 8 word blocks

A = 32, C = 32 KiB = \(2^{15}\) B, N = 4, K = 8 words = 32 B

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>I</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>21</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>(B)</td>
<td><strong>19</strong></td>
<td><strong>8</strong></td>
<td><strong>5</strong></td>
</tr>
<tr>
<td>(C)</td>
<td>19</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>(D)</td>
<td>17</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

O = \(\log_2(K)\) = 5 bits
C/K = \(2^{10}\) slots
C/K/N = \(2^8\) sets
I = \(\log_2(C/K/N)\) = 8 bits
T = A – I – O = 19 bits
Summary

- Set associativity determines flexibility of block placement
  - Fully associative: blocks can go anywhere
  - Direct-mapped: blocks go in one specific location
  - N-way: cache split into sets, each of which have $n$ slots to place memory blocks