Cache Memories
Direct Mapping

• Direct mapping:
  – A memory value can only be placed at a single corresponding location in the cache
Direct-Mapped:
One cache location for each address

Address

Set field selects line in cache

Cache

Tag Set Offset

Tag

Data

Compare

Hit

Requested Data
Fully Associative Mapping

- Fully-associative mapping:
  - A memory value can be placed anywhere in the cache
Fully-Associative: Anything Can Go Anywhere

Check each line in parallel

Select line to return based on which line hit

Requested Data
Set Associative Mapping (2-Way)

- Set-associative mapping:
  - A memory value can be placed in any location of a set in the cache
Compromise: Set-Associative Caches

Check all ways of a set in parallel

Hit?

Requested Data

Address
Tag Set Offset

Way

Select Set In Cache

Set

Cache

Compare

Hit?

Requested Data
Three Cs (Cache Miss Terms)

- Compulsory Misses:
  - Cold start misses (caches do not have valid data at the start of the program)
Three Cs (Cache Miss Terms)

• Capacity Misses:
  – Increase cache size

![Diagram showing processor and cache with hexadecimal values]
Three Cs (Cache Miss Terms)

- Conflict Misses:
  - Increase cache size and/or associativity.
  - Associative caches reduce conflict misses
A 4-way set associative cache
Cache Organization -- Recap

- A typical cache has three dimensions

```
| tag | index | block offset |
```

Number of sets (cache size)

Blocks/set (associativity)

Bytes/block (block size)
Cache Parameters

Cache size = Number of sets * block size * associativity

128 blocks, 32-byte blocks, direct mapped, size = ?

128 KB cache, 64-byte blocks, 512 sets, associativity = ?
Putting it all together
64 KB cache, direct-mapped, 32-byte cache block

64 KB / 32 bytes = 2 K cache blocks/sets

hit/miss

word offset
A set associative cache

32 KB cache, 2-way set-associative, 16-byte blocks

This picture doesn’t show the “most recent” bit (need one bit per set)
A cache has a capacity of 32 KB and 256-byte lines. On a machine with a 32-bit virtual address space, how many bits long are the tag, set, and offset fields for

- A direct-mapped implementation?
- A four-way set-associative implementation?
- A fully-associative implementation?
A cache has a capacity of 32 KB and 256-byte in a set. On a machine with a 32-bit address space, how many bits long are the tag, set, and offset fields for

• A direct-mapped implementation?
• A four-way set-associative implementation?
• A eight-way set-associative implementation?
Another example

- 16 KB, 4-way set-associative cache, 32-bit address, byte-addressable memory, 32-byte cache blocks/lines
- how many tag bits?
- Where would you find the word at address 0x200356A4?
Putting it all together

64 KB cache, direct-mapped, 32-byte cache block

64 KB / 32 bytes = 2 K cache blocks/sets

hit/miss

64 KB cache, direct-mapped, 32-byte cache block

64 KB / 32 bytes = 2 K cache blocks/sets

hit/miss
PIRW

• Placement
• Identification
• Replacement
• Writes
Replacement Policies for Associative Caches

- **Least-Recently-Used (LRU):** Evict the line that has been least recently referenced
  - Need to keep track of order that lines in a set have been referenced
  - Overhead to do this gets worse as associativity increases
- **Random:** Just pick one at random
  - Easy to implement
  - Slightly lower hit rates than LRU on average
- **Not-Most-Recently-Used:** Track which line in a set was referenced most recently, pick randomly from the others
  - Compromise in both hit rate and implementation difficulty
- Virtual memories use similar policies, but are willing to spend more effort to improve hit rate
LRU: Precise Tracking

- Precise LRU tracking requires a stack for each set
  - When a block is accessed by the processor, check if its name is in the stack. If so, remove it from the stack. Push the name of block at the top of the stack
  - The position (depth) of a block name in the stack gives the relative recency of access to this block compared to others
  - For a 4-way set associative cache with blocks A, B, C, D.
    - Assume a sequence of accesses C, D, A, B, A, C, B, D
    - Assume that the stack is initially empty, the configuration of the stack after each access would be [C,-,-,-], [D, C,-,-], [A, D, C,-], [B, A, D, C], [A, B, D, C], [C, A, B, D], [B, C, A, D], [D, B, C, A]
    - The one on the right most position (bottom of the stack) is the least recently used
    - Each name takes two bits, so the stack is an 8-bit register with associated logic ($\text{Slog}_2 S$ where $S$ is associativity)
LRU Example

Access C
- MRU: A, MRU-1: B, LRU+1: C, LRU: D

Access D
- MRU: D, MRU-1: C, LRU+1: A, LRU: B

Access E
- MRU: E, MRU-1: D, LRU+1: C, LRU: A

Access C
- MRU: C, MRU-1: E, LRU+1: D, LRU: A

Access G
- MRU: G, MRU-1: C, LRU+1: E, LRU: D

MISS, replacement needed

MISS, replacement needed
LRU policy increases cache access times
Additional hardware bits needed for LRU state machine
LRU: Approximate Tracking (Pseudo-LRU)

• LRU stacks expensive to implement at high ‘S’
• Most caches use approximate LRU
  – A popular approach uses S-1 bits for an S-way cache
    • The blocks are hierarchically divided into a binary tree
    • At each level of the tree, one bit is used to track the least recently used
  – For a 4-way set associate cache, blocks are first divided into two halves, each half has two blocks
    • The 1st bit tracks the more recently used half
    • The 2nd bit (3rd) tracks the more recently block in the first (second) half
    • The one to replace is the less recently used block in the less recently used half
• used in the CPU cache of many commercial processors
Pseudo LRU Algorithm (4-way SA)

- Tree-based
  - O(N): 3 bits for 4-way
  - Cache ways are the leaves of the tree
  - Combine ways as we proceed towards the root of the tree
Pseudo LRU Algorithm

Less hardware than LRU & Faster than LRU

LRU update algorithm

<table>
<thead>
<tr>
<th>Way hit</th>
<th>CD</th>
<th>AB</th>
<th>AB/CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Way A</td>
<td>---</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Way B</td>
<td>---</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Way C</td>
<td>0</td>
<td>---</td>
<td>1</td>
</tr>
<tr>
<td>Way D</td>
<td>1</td>
<td>---</td>
<td>1</td>
</tr>
</tbody>
</table>

Replacement Decision

<table>
<thead>
<tr>
<th>L2</th>
<th>L1</th>
<th>L0</th>
<th>Way to replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>Way A</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Way B</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>Way C</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>Way D</td>
</tr>
</tbody>
</table>

L2L1L0 = 000, there is a hit in Way B, what is the new updated L2L1L0?

L2L1L0 = 010, a way needs to be replaced, which way would be chosen?
LRU Algorithms

• True LRU
  – Expensive in terms of speed and hardware
  – Need to remember the order in which all N lines were last accessed
  – N! scenarios – O(log N!) \approx O(N \log N) LRU bits
    • 2-ways \rightarrow AB BA = 2 = 2!
    • 3-ways \rightarrow ABC ACB BAC BCA CAB CBA = 6 = 3!

• Pseudo LRU: O(N)
  – Approximates LRU policy with a binary tree
How about an approximate LRU for 8-way?
A Sample Replacement Policy Question

• A byte-addressable computer has a small 32-byte cache. Assume 4-byte cache lines. When a given program is executed, the processor reads data from the following sequence of hex addresses: 200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4. This pattern is repeated four times.

  – (a) Show the contents of the cache at the end of each pass throughout this loop if a direct-mapped cache is used. Compute the hit rate for this example. Assume that the cache is initially empty.

  – (b) Repeat part (a) for a fully-associative cache that uses the LRU-replacement algorithm.

  – (c) Repeat part (a) for a fully-associative cache that uses the approximate LRU replacement algorithm.
You should be able to complete the answer

• 200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4 (Hex Address)
• 4-byte blocks so block address sequence in Hex is
  – 80, 81, 82, 83, BD, BC, 80, 81, 86, 87, ...
• Direct mapped, 8 blocks
  – 1000 0000, 1000 0001, 1000 0010, 1000 0011, 1010 1101, 1010 1100, 1000 0000, 1000 0001, 1000 0110, 1000 0111,
  – For direct map, there is no room for policy
  – Complete your answer...
• For fully associative cache, it is really an 8-way, 1-set cache, all block address bits are used
  – Complete your answer...
Another Question

• Consider a 128 byte 2-way set associative write-back cache with 16 byte blocks. Assume LRU replacement and that dirty bits are used to avoid writing back clean blocks. Complete the table below for a sequence of memory references (occurring from left to right).

• Address (in decimal): 064 032 064 000 112 064 128 048 240 000 read/write: r r r r w w r r r w

• Assume that cache starts empty
You should complete the answer

16 byte blocks, 4 LSB byte offset

064 032 064 000 112 064 128 048 240 000

There are 4 sets with 2 blocks each

<table>
<thead>
<tr>
<th>Block Addr.</th>
<th>4</th>
<th>2</th>
<th>4</th>
<th>0</th>
<th>7</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>Set#</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>H/M</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write back?</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write-through Policy
Write-back Policy

Processor

0x9ABC

Cache

0x5678

Memory

?????

Write miss
On Write Miss

• **Write allocate**
  – The line is allocated on a write miss, followed by the write hit actions above.
  – Write misses first act like read misses
  – Better performance if data referenced again

• **No write allocate**
  – Write misses do not interfere cache
  – Line is only modified in the lower level memory
  – Mostly use with write-through cache
  – Simpler write hardware
  – May be better for small caches if written data won’t be read again soon
Reducing Miss Rate

- Enlarge cache
- If cache size is fixed
  - Increase associativity and/or line size (does this always work?)

Increasing cache pollution
An example problem

• Consider a 128 byte 2-way set associative write-back cache with 16 byte blocks. Assume LRU replacement and that dirty bits are used to avoid writing back clean blocks. Complete the table below for a sequence of memory references (occurring from left to right).

• Address: 064 032 064 000 112 064 128 048 240 000
  read/write:r r r r w w r r r w
• set#: 02
• Tag: 10
• hit/miss: miss
• write back? no
Write-Allocate vs. Write-no-Allocate

Two options if a store causes a miss in the cache

- **Write-allocate**: Fetch line into cache, then perform the write in the cache
- **Write-no-allocate**: Pass the write through to the main memory, don’t bring the line into the cache

**Tradeoffs**
- WA has Better performance if data referenced again before it is evicted
- WNA has
  - Simpler write hardware
  - May be better for small caches if written data won’t be read again soon

Which makes more sense for writeback/write-through?
Instruction and Data Caches

CPU

References to Instructions

Instruction Cache (I-Cache)

References to Data

Data Cache (D-Cache)

Main Memory
Cache Performance Example

• Given
  – I-cache miss rate = 2%
  – D-cache miss rate = 4%
  – Miss penalty = 100 cycles
  – Base CPI (ideal cache) = 2
  – Load & stores are 36% of instructions

What is the actual CPI?
# 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong> (per core)</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td></td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td><strong>L2 unified cache</strong> (per core)</td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>L3 unified cache (shared)</strong></td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
</tbody>
</table>

**n/a:** data not available