What to do on a write hit?

- **Write-through**
  - update the word in cache block and corresponding word in memory
- **Write-back**
  - update word in cache block
  - allow memory word to be “stale”
    - add ‘dirty’ bit to each block indicating that memory needs to be updated when block is replaced
    - OS flushes cache before I/O...

Performance trade-offs?

#### Block Size Tradeoff (1/3)
- Benefits of Larger Block Size
  - **Spatial Locality**: if we access a given word, we’re likely to access other nearby words soon
  - Very applicable with Stored-Program Concept: if we execute a given instruction, it’s likely that we’ll execute the next few as well
  - Works nicely in sequential array accesses too

#### Block Size Tradeoff (2/3)
- Drawbacks of Larger Block Size
  - Larger block size means larger miss penalty
  - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up
  - In general, minimize average memory access time (AMAT)
    - Hit Time
    - Miss Penalty x Miss Rate

#### Block Size Tradeoff (3/3)
- **Hit Time** = time to find and retrieve data from current level cache
- **Miss Penalty** = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- **Hit Rate** = % of requests that are found in current level cache
- **Miss Rate** = 1 - Hit Rate

#### Extreme Example: One Big Block
- **Cache Size** = 4 bytes
- **Block Size** = 4 bytes
  - Only **ONE** entry (row) in the cache!
  - If item accessed, likely accessed again soon
    - But unlikely will be accessed again immediately!
  - The next access will likely to be a miss again
  - Continuously loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

100 Gig Eth?! ➞ Yesterday Infinera demoed a system that took a 100 GigE signal, broke it up into ten 10GigE signals and sent it over existing optical networks. Fast!!!

100 Gig Eth?!! ➞ gigaom.com/2006/11/14/100gbe
Block Size Tradeoff Conclusions

Miss Penalty
 Miss Rate
Exploits Spatial Locality
Fewer blocks: compromises
temporal locality
Average Access
Time
Increased Miss Penalty
& Miss Rate

Types of Cache Misses (1/2)

• “Three Cs” Model of Misses

• 1st C: Compulsory Misses
  - occur when a program is first started
  - cache does not contain any of that program’s data yet, so misses are bound
to occur
  - can’t be avoided easily, so won’t focus
  on these in this course

Types of Cache Misses (2/2)

• 2nd C: Conflict Misses
  - miss that occurs because two distinct memory
    addresses map to the same cache location
  - two blocks (which happen to map to the same
    location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
• Dealing with Conflict Misses
  - Solution 1: Make the cache size bigger
    - fails at some point
  - Solution 2: Multiple distinct blocks can fit in the
    same cache Index?

Fully Associative Cache (1/3)

• Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
• What does this mean?
  - no “rows”: any block can go anywhere in
    the cache
  - must compare with all tags in entire cache
to see if data is there

Fully Associative Cache (2/3)

• Fully Associative Cache (e.g., 32 B block)
  - compare tags in parallel

<table>
<thead>
<tr>
<th>Cache Tag (32 bits long)</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>Cache Tag</td>
<td>Valid</td>
</tr>
<tr>
<td>Cache Data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Valid</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Fully Associative Cache (3/3)

• Benefit of Fully Assoc Cache
  - No Conflict Misses (since data can go
    anywhere)
• Drawbacks of Fully Assoc Cache
  - Need hardware comparator for every
    single entry: if we have a 64KB of data in
    cache with 4B entries, we need 16K
    comparators: infeasible
Third Type of Cache Miss

• **Capacity Misses**
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea
  - This is the primary type of miss for Fully Associative caches.

N-Way Set Associative Cache (1/3)

• Memory address fields:
  - **Tag**: same as before
  - **Offset**: same as before
  - **Index**: points us to the correct “row” (called a set in this case)

• So what’s the difference?
  - each set contains multiple blocks
  - once we’ve found correct set, must compare with all tags in that set to find our data

Associative Cache Example

- Here’s a simple 2 way set associative cache.

N-Way Set Associative Cache (2/3)

• Basic Idea
  - cache is direct-mapped w/respect to sets
  - each set is fully associative
  - basically N direct-mapped caches working in parallel: each has its own valid bit and data

• Given memory address:
  - Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.

N-Way Set Associative Cache (3/3)

• What’s so great about this?
  - even a 2-way set assoc cache avoids a lot of conflict misses
  - hardware cost isn’t that bad: only need N comparators

• In fact, for a cache with M blocks,
  - it’s Direct-Mapped if it’s 1-way set assoc
  - it’s Fully Assoc if it’s M-way set assoc
  - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit
Block Replacement Policy

• Direct-Mapped Cache: index completely specifies position which position a block can go in on a miss
• N-Way Set Assoc: index specifies a set, but block can occupy any position within the set on a miss
• Fully Associative: block can be written into any position

Question: if we have the choice, where should we write an incoming block?

• If there are any locations with valid bit off (empty), then usually write the new block into the first one.
• If all possible locations already have a valid block, we must pick a replacement policy: rule by which we determine which block gets “cached out” on a miss.

Block Replacement Policy: LRU

• LRU (Least Recently Used)
  • Idea: cache out block which has been accessed (read or write) least recently
  • Pro: temporal locality ⇒ recent past use implies likely future use: in fact, this is a very effective policy
  • Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

• We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):
  0, 2, 0, 1, 4, 0, 2, 3, 5, 4

How many hits and how many misses will there be for the LRU block replacement policy?

Block Replacement Example: LRU

• Addresses 0, 2, 0, 1, 4, 0, ...

<table>
<thead>
<tr>
<th>Location</th>
<th>LRU</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

How many hits and how many misses will there be for the LRU block replacement policy?

Big Idea

• How to choose between associativity, block size, replacement & write policy?
• Design against a performance model
  • Minimize: \( \text{Average Memory Access Time} = \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate} \)
  • influenced by technology & program behavior
• Create the illusion of a memory that is large, cheap, and fast - on average

Improving Miss Penalty

• When caches first became popular, Miss Penalty ~ 10 processor clock cycles
• Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM ⇒ 200 processor clock cycles!

Solution: another cache between memory and the processor cache: Second Level (L2) Cache
Analyzing Multi-level cache hierarchy

**Diagram:**
- L1 hit time
- L2 hit time
- L1 Miss Rate
- L2 Miss Rate
- L1 Miss Penalty
- L2 Miss Penalty

**Equations:**
- Avg Mem Access Time = L1 Hit Time + L1 Miss Rate * L1 Miss Penalty
- L1 Miss Penalty = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty
- Avg Mem Access Time = L1 Hit Time + L1 Miss Rate * (L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

And in Conclusion...
- We’ve discussed memory caching in detail. Caching in general shows up over and over in computer systems:
  - Filesystem cache
  - Web page cache
  - Game databases/tablebases
  - Software memoization
  - Others?
- Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.
- Cache design choices:
  - Write through v. write back
  - Size of cache: speed v. capacity
  - Direct-mapped v. associative
  - For N-way set assoc: choice of N
  - Block replacement policy
  - 2nd level cache?
  - 3rd level cache?
- Use performance model to pick between choices, depending on programs, technology, budget...

Example
- Assume
  - Hit Time = 1 cycle
  - Miss rate = 5%
  - Miss penalty = 20 cycles
  - Calculate AMAT...
- Avg mem access time
  \[= 1 + 0.05 \times 20\]
  \[= 1 + 1 \text{ cycles}\]
  \[= 2 \text{ cycles}\]

Ways to reduce miss rate
- Larger cache
  - Limited by cost and technology
  - Hit time of first level cache < cycle time (bigger caches are slower)
- More places in the cache to put each block of memory – associativity
  - Fully-associative
    - Any block any line
  - N-way set associated
    - N places for each block
  - Direct map: N=1

Typical Scale
- L1
  - Size: tens of KB
  - Hit time: complete in one clock cycle
  - Miss rates: 1-5%
- L2:
  - Size: hundreds of KB
  - Hit time: few clock cycles
  - Miss rates: 10-20%
- L2 miss rate is fraction of L1 misses that also miss in L2
  - Why so high?
Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles
- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35 = 2.75 cycles

Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
- Avg mem access time = 1 + 0.05 x 200 = 11 cycles

4x faster with L2 cache! (2.75 vs. 11)

An actual CPU – Early PowerPC

- Cache
  - 32 KByte Instructions and 32 KByte Data L1 caches
  - External L2 Cache interface with integrated controller and cache tags, supports up to 1 MByte external L2 cache
  - Dual Memory Management Units (MMU) with Translation Lookaside Buffers (TLB)
- Pipelining
  - Superscalar (3 ins/cycle)
  - 8 execution units (2 integer and 1 double precision IEEE floating point)

An Actual CPU – Pentium M

- New Microarchitecture
- 32K I\$ 32K D\$
- Advanced Pentium M microarchitecture
- Enhanced performance
- 77 Mhz for Pentium M