**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**

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>00000000</td>
</tr>
</tbody>
</table>

- Cache Size = 4 bytes  Block Size = 4 bytes
  - Only ONE entry 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
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

**Block Size Tradeoff Conclusions**

- Exploits Spatial Locality
  - Fewer blocks: compromises temporal locality
- Increased Miss Penalty & Miss Rate

- Average Memory Access Time
  - Block Size
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 (27 bits long)</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Tag Valid Cache Data</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>B 31 ...</td>
<td>B 1</td>
</tr>
<tr>
<td>0</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 (1/4)

- 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

N-Way Set Associative Cache (2/4)

- Summary:
  - 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

N-Way Set Associative Cache (3/4)

- 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 (4/4)

- 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

Associative Cache Example

- Recall this is how a simple direct mapped cache looked.
- This is also a 1-way set-associative cache!

Associative Cache Example

- Here’s a simple 2 way set associative cache.
Block Replacement Policy (1/2)

- **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?

Block Replacement Policy (2/2)

- **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

<table>
<thead>
<tr>
<th>Loc 0</th>
<th>Loc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: miss, bring into set 0 (loc 0)</td>
<td>0 1</td>
</tr>
<tr>
<td>2: miss, bring into set 0 (loc 1)</td>
<td>0 2</td>
</tr>
<tr>
<td>0: hit</td>
<td>0 2</td>
</tr>
<tr>
<td>1: miss, bring into set 1 (loc 0)</td>
<td>0 2</td>
</tr>
<tr>
<td>4: miss, bring into set 0 (loc 1, replace 2)</td>
<td>0 4</td>
</tr>
<tr>
<td>0: hit</td>
<td>0 4</td>
</tr>
</tbody>
</table>

Big Idea

- **How to choose between associativity, block size, replacement policy?**
- **Design against a performance model**
  - Minimize: **Average Memory Access Time**
    - Hit Time
    - Miss Penalty x Miss Rate
  - Influenced by technology & program behavior
  - Note: **Hit Time encompasses Hit Rate!!!**
- **Create the illusion of a memory that is large, cheap, and fast - on average**
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
• 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

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

\[
\text{Avg Mem Access Time} = \ L1 \text{ Hit Time} + L1 \text{ Miss Rate} \times L1 \text{ Miss Penalty}
\]

\[
\text{L2 Miss Penalty} = L2 \text{ Hit Time} + L2 \text{ Miss Rate} \times L2 \text{ Miss Penalty}
\]

\[
\text{Avg Mem Access Time} = L1 \text{ Hit Time} + L1 \text{ Miss Rate} \times (L2 \text{ Hit Time} + L2 \text{ Miss Rate} \times L2 \text{ Miss Penalty})
\]

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 = \( 5 + 0.15 \times 200 = 35 \)
• Avg mem access time = \( 1 + 0.05 \times 35 = 2.75 \text{ cycles} \)
**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)

---

**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?

---

**Peer Instructions**

1. In the last 10 years, the gap between the access time of DRAMs & the cycle time of processors has decreased. (i.e., is closing)
2. A 2-way set-associative cache can be outperformed by a direct-mapped cache.
3. Larger block size ⇒ lower miss rate

---

**Generalized Caching**

- We’ve discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - Filesystem cache
  - Web page cache
  - Game Theory 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.

---

**An actual CPU -- Early PowerPC**

- Cache
  - 32 KiByte Instructions and 32 KiByte 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 inst/cycle)
  - 6 execution units (2 integer and 1 double precision IEEE floating point)