Dealing the Interference

- By bad luck or pathological happenstance a particular line in the cache may be highly contended.
- How can we deal with this?
Interfering Code.

int foo[129]; // 4*129 = 516 bytes
int bar[129]; // Assume the compiler aligns these at 512 byte boundaries

while(1) {
    for (i = 0; i < 129; i++) {
        s += foo[i]*bar[i];
    }
}

- Assume a 1KB (0x400 byte) cache.
- Foo and Bar map into exactly the same part of the cache.
- Is the miss rate for this code going to be high or low?
- What would we like the miss rate to be?
- Foo and Bar should both (almost) fit in the cache!
Associativity

- (set) Associativity means providing more than one place for a cache line to live.
- The level of associativity is the number of possible locations
  - 2-way set associative
  - 4-way set associative
- One group of lines corresponds to each index
  - it is called a “set”
- Each line in a set is called a “way”
Associativity

<table>
<thead>
<tr>
<th>dirty</th>
<th>valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 0

Way 1
New Cache Geometry Calculations

Addresses break down into: tag, index, and offset.
How they break down depends on the “cache geometry”

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)
- Associativity = W

- Index bits = \( \log_2(L/W) \)
- Offset bits = \( \log_2(B) \)
- Tag bits = A - (index bits + offset bits)
Practice

- 32KB, 2048 Lines, 4-way associative.

- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits: 19
- Offset bits: 4
Fully Associative and Direct Mapped Caches

- At one extreme, a cache can have one, large set. The cache is then **fully associative**.
- At the other, it can have one cache line per set. Then it is **direct mapped**.
Eviction in Associative caches

- We must choose which line in a set to evict if we have associativity.
- How we make the choice is called the cache eviction policy.
  - Random -- always a choice worth considering. Hard to implement true randomness.
  - Least recently used (LRU) -- evict the line that was last used the longest time ago.
  - Prefer clean -- try to evict clean lines to avoid the write back.
  - Farthest future use -- evict the line whose next access is farthest in the future. This is provably optimal. It is also impossible to implement.
The Cost of Associativity

- Increased associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
  - The fastest way is to use a “content addressable memory” They embed comparators in the memory array. -- try instantiating one in Xlinix.

- This limits associativity L1 caches to 2-8.
- Larger, slower caches can be more associative.
- Example: Nehalem
  - 8-way L1
  - 16-way L2 and L3.
- Core 2’s L2 was 24-way
Increasing Bandwidth

- A single, standard cache can service only one operation at time.
- We would like to have more bandwidth, especially in modern multi-issue processors.
- There are two choices
  - Extra ports
  - Banking
Extra Ports

• Pros: Uniformly supports multiple accesses
  • Any N addresses can be accessed in parallel.
• Costly in terms of area.
  • Remember: SRAM size increases quadratically with the number of ports
Banking

- Multiple, independent caches, each assigned one part of the address space (use some bits of the address)
- Pros: Efficient in terms of area. Four banks of size \( \frac{N}{4} \) are only a bit bigger than one cache of size \( N \).
- Cons: Only one access per bank. If you are unlucky, multiple accesses will target the same bank (structural hazard).