Caching In Depth
Today

- Quiz
- Design choices in cache architecture
Basic Cache Organization

• Some number of cache lines each with
  • Dirty bit -- does this data match what is in memory
  • Valid -- does this mean anything at all?
  • Tag -- The high order bits of the address
  • Data -- The program’s data

• Note that the index of the line, combined with the tag, uniquely identify one cache line’s worth of memory
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)

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

• 1024 cache lines. 32 Bytes per line.
• Index bits:
• Tag bits:
• off set bits:
• 1024 cache lines. 32 Bytes per line.
• Index bits: 10
• Tag bits:
• off set bits:
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits:
- off set bits: 5
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits: 17
- Offset bits: 5
Practice

• 32KB cache.
• 64byte lines.

• Index
• Offset
• Tag
Practice

- 32KB cache.
- 64byte lines.
- Index
- Offset
- Tag
Practice

- 32KB cache.
- 64byte lines.

- Index  9
- Offset
- Tag  17
Practice

• 32KB cache.
• 64byte lines.

• Index 9
• Offset 6
• Tag 17
The basic read algorithm

{tag, index, offset} = address;
if (isRead) {
    if (tags[index] == tag) {
        return data[index];
    } else {
        l = chooseLine(...);
        if (l is dirty) {
            WriteBack(l);
        }
        Load address into line l;
        return data[l];
    }
}
The basic read algorithm

\[
\text{\{tag, index, offset\} = address;}
\]

\[
\text{if (isRead) { }
\]

\[
\text{if (tags[index] == tag) {}
\]

\[
\text{return data[index];}
\]

\[
\text{} \text{else { }
\]

\[
\text{l = chooseLine(...);}
\]

\[
\text{if (l is dirty) {}
\]

\[
\text{WriteBack(l);}
\]

\[
\text{}}
\]

\[
\text{Load address into line l;}
\]

\[
\text{return data[l];}
\]

\[
\}
\]

**Which line to evict?**
The basic write algorithm

{tag, index, offset} = address;
if (isWrite) {
    if (tags[index] == tag) {
        data[index] = data; // Should we just update locally?
        dirty[index] = true;
    } else {
        l = chooseLine(...); // maybe no line?
        if (l is dirty) {
            WriteBack(l);
        }
        if (l exists) {
            data[l] = data;
        }
    }
}
The basic write algorithm

\{tag, index, offset\} = address;
if (isWrite) {
  if (tags[index] == tag) {
    data[index] = data; // Should we just update locally?
    dirty[index] = true;
  } else {
    l = chooseLine(...); // maybe no line?
    if (l is dirty) {
      WriteBack(l);
    }
    if (l exists) {
      data[l] = data;
    }
  }
}

Where to write data?
The basic write algorithm

{tag, index, offset} = address;
if (isWrite) {
    if (tags[index] == tag) {
        data[index] = data; // Should we just update locally?
        dirty[index] = true;
    } else {
        l = chooseLine(...); // maybe no line?
        if (l is dirty) {
            WriteBack(l);
        }
        if (l exists) {
            data[l] = data;
        }
    }
}
The basic write algorithm

\{tag, index, offset\} = address;
if (isWrite) {
    if (tags[index] == tag) {
        data[index] = data; // Should we just update locally?
        dirty[index] = true;
    } else {
        l = chooseLine(...); // maybe no line?
        if (l is dirty) {
            WriteBack(l);
        }
        if (l exists) {
            data[l] = data;
        }
    }
} else {
    l = chooseLine(...); // maybe no line?
    if (l exists) {
        data[l] = data;
    }
}

Where to write data?
Should we evict something?
What should we evict?
Write Design Choices

• Remember all decisions are only for this cache. The lower levels of the hierarchy might make different decisions.

• Where to write data?
  • Write-through -- Writes to this cache and the next lower level of the hierarchy.
  • No-write-through -- Writes only affect this level

• Should we evict anything?
  • Write-allocate -- bring the modified line into the cache, then modify it.
  • No-write-allocate -- Update the cache line where you find it in the hierarchy. Do not bring it “closer”

• What to evict?
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?
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>

Set 0

Way 0

Set 1

Way 1

Set 2

Set 3
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:
• Sets:
• Index bits:
• Tag bits:
• Offset bits:
Practice

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

• Line size: \(16B\)
• Sets:
• Index bits:
• Tag bits:
• Offset bits:
Practice

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

- Line size: 16B
- Sets: 512
- Index bits:
- Tag bits:
- Offset bits:
Practice

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

• Line size: 16B
• Sets: 512
• Index bits: 9
• Tag bits:
• Offset bits:
Practice

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

- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits:
- Offset bits: 4
Practice

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

• Line size: 16B
• Sets: 512
• Index bits: 9
• Tag bits: 19
• Offset bits: 4
Full Associativity

• In the limit, a cache can have one, large set.
• The cache is then fully associative
• A one-way associative cache is also called --
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 difficult 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 2-4. In L2s to make 16 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 $N/4$ are only a bit bigger than one cache of size $N$.
- Cons: Only one access per bank. If you are unlucky you don’t get the extra.