This Unit: Caches and Memory Hierarchies

- Memory hierarchy
  - Basic concepts
- SRAM technology
  - Transistors and circuits
- Cache organization
  - ABCs
  - CAM (content associative memory)
  - Classifying misses
  - Two optimizations
  - Writing into a cache
- Some example calculations
Basic Cache Structure

- Basic cache: array of block frames
  - Example: 4KB cache made up of 1024 4B frames
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 4B blocks $\rightarrow$ 2 LS bits locate byte within block
    - These are called offset bits
  - 1024 frames $\rightarrow$ next 10 bits find frame
    - These are the index bits
  - Note: nothing says index must be these bits
  - But these work best (think about why)
Basic Cache Structure

- Each frame can hold one of $2^{20}$ blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - If (tag matches && valid bit set)
    then Hit $\rightarrow$ data is good
  Else Miss $\rightarrow$ data is no good, wait
Calculating Tag Size

• “4KB cache” means cache holds 4KB of data
  • Called capacity
  • Tag storage is considered overhead (not included in capacity)

• Calculate tag overhead of 4KB cache with 1024 4B frames
  • Not including valid bits
  • 4B frames → 2-bit offset
  • 1024 frames → 10-bit index
  • 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  • 20-bit tag * 1024 frames = 20Kb tags = 2.5KB tags
  • 63% overhead
Measuring Cache Performance

- Ultimate metric is $t_{\text{avg}}$
  - Cache capacity roughly determines $t_{\text{hit}}$
  - Lower-level memory structures determine $t_{\text{miss}}$
- Measure $\%_{\text{miss}}$
  - Hardware performance counters (Pentium, Sun, etc.)
  - Simulation (write a program that mimics behavior)
  - Hand simulation (next slide)
- $\%_{\text{miss}}$ depends on program that is running
  - Why?
## Cache Performance Simulation

- **Parameters:** 8-bit addresses, 32B cache, 4B blocks
  - Addresses initially in cache: 0, 4, 8, 12, 16, 20, 24, 28
  - To find location in cache, do mod32 arithmetic (why 32?)

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 4, 8, 12, 16, 20, 24, 28</td>
<td>200 (200%32=8)</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 200, 12, 16, 20, 24, 28</td>
<td>204 (204%32=12)</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 200, 204, 16, 20, 24, 28</td>
<td>144 (144%32=16)</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 200, 204, 144, 20, 24, 28</td>
<td>6</td>
<td>Hit</td>
</tr>
<tr>
<td>0, 4, 200, 204, 144, 20, 24, 28</td>
<td>8</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 8, 204, 144, 20, 24, 28</td>
<td>12</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 8, 12, 144, 20, 24, 28</td>
<td>20</td>
<td>Hit</td>
</tr>
<tr>
<td>0, 4, 8, 12, 144, 20, 24, 28</td>
<td>16</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 8, 12, 16, 20, 24, 28</td>
<td>144</td>
<td>Miss</td>
</tr>
<tr>
<td>0, 4, 8, 12, 144, 20, 24, 28</td>
<td>200</td>
<td>Miss</td>
</tr>
</tbody>
</table>
Block Size

- Given capacity, manipulate $\%_{\text{miss}}$ by changing organization
- One option: increase **block size**
  - + Exploit **spatial locality**
    - • Caveat: works only up to a point
  - + Reduce tag overhead
- • Notice tag/index/offset bits
Calculating Tag Size

- Calculate tag overhead of 4KB cache with 512 8B frames
  - Not including valid bits
  - 8B frames → 3-bit offset
  - 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - 20-bit tag * 512 frames = 10Kb tags = 1.25KB tags
    + 32% overhead
      + Less tag overhead with larger blocks
Cache Performance Simulation

- Parameters: 8-bit addresses, 32B cache, **8B blocks**
  - Addresses in base4 ("nibble") notation
  - Initial contents: 0000(0010), 0020(0030), 0100(0110), 0120(0130)

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000(0010), 0020(0030), 0100(0110), 0120(0130)</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), <strong>3020(3030)</strong>, 0100(0110), 0120(0130)</td>
<td>3030</td>
<td><strong>Hit (spatial locality!)</strong></td>
</tr>
<tr>
<td>0000(0010), 3020(3030), 0100(0110), 0120(0130)</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), 3020(3030), <strong>2100(2110)</strong>, 0120(0130)</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000(0010), 3020(3030), 2100(2110), 0120(0130)</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), <strong>0020(0030)</strong>, 2100(2110), 0120(0130)</td>
<td>0030</td>
<td><strong>Hit (spatial locality!)</strong></td>
</tr>
<tr>
<td>0000(0010), 0020(0030), 2100(2110), 0120(0130)</td>
<td>0110</td>
<td><strong>Miss (conflict)</strong></td>
</tr>
<tr>
<td>0000(0010), 0020(0030), <strong>0100(0110)</strong>, 0120(0130)</td>
<td>0100</td>
<td><strong>Hit (spatial locality!)</strong></td>
</tr>
<tr>
<td>0000(0010), 0020(0030), 0100(0110), 0120(0130)</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000(0010), 0020(0030), <strong>2100(2110)</strong>, 0120(0130)</td>
<td>3020</td>
<td>Miss</td>
</tr>
</tbody>
</table>
Effect of Block Size

- Increasing block size has two effects (one good, one bad)
  + Spatial prefetching
    - For blocks with adjacent addresses
    - Turns miss/miss pairs into miss(hit) pairs
    - Example from previous slide: 3020, 3030
  - Conflicts
    - For blocks with non-adjacent addresses (but adjacent frames)
    - Turns hits into misses by disallowing simultaneous residence
    - Example: 2100, 0110
  - Both effects always present to some degree
  - Spatial prefetching dominates initially (until 64–128B)
  - Interference dominates afterwards
  - Optimal block size is 32–128B (varies across programs)
Conflicts

• What about pairs like 3030/0030, 0100/2100?
  • These will conflict in any size cache (regardless of block size)
  • Will keep generating misses

• Can we allow pairs like these to simultaneously reside?
  • Yes, but we have to reorganize cache to do so

---

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130</td>
<td><strong>3030</strong></td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, <strong>3030</strong>, 0100, 0110, 0120, 0130</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, 0030, 2100, 0110, 0120, 0130</td>
<td><strong>0030</strong></td>
<td>Miss</td>
</tr>
<tr>
<td>0000, 0010, 0020, <strong>0030</strong>, 2100, 0110, 0120, 0130</td>
<td>0110</td>
<td>Hit</td>
</tr>
</tbody>
</table>
Set-Ass ociativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called **sets**
  - Each frame in set called a **way**
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

+ Reduces conflicts
- Increases $t_{hit}$: additional mux

[Diagram showing set-associative cache with data, CPU address, and hit/miss]
Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - Any (match && valid bit)?
    - Then Hit
    - Else Miss
  - Notice tag/index/offset bits

[Diagram of memory structure and lookup algorithm with addresses and bit representations]

© 2009 Daniel J. Sorin from Roth and Lebeck
Cache Performance Simulation

- Parameters: 32B cache, 4B blocks, **2-way set-associative**
- Initial contents: 0000, 0010, 0020, 0030, 0100, 0110, 0120, 0130

<table>
<thead>
<tr>
<th>Cache contents</th>
<th>Address</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0000,0100], [0010,0110], [0020,0120], [0030,0130]</td>
<td>3020</td>
<td>Miss</td>
</tr>
<tr>
<td>[0000,0100], [0010,0110], [0120,3020], [0030,0130]</td>
<td>3030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0000,0100], [0010,0110], [0120,3020], [0130,3030]</td>
<td>2100</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [0120,3020], [0130,3030]</td>
<td>0012</td>
<td>Hit</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [0120,3020], [0130,3030]</td>
<td>0020</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [0130,3030]</td>
<td>0030</td>
<td>Miss</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>0110</td>
<td>Hit</td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>0100</td>
<td><strong>Hit (avoid conflict)</strong></td>
</tr>
<tr>
<td>[2100,0100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>2100</td>
<td><strong>Hit (avoid conflict)</strong></td>
</tr>
<tr>
<td>[0100,2100], [0010,0110], [3020,0020], [3030,0030]</td>
<td>3020</td>
<td><strong>Hit (avoid conflict)</strong></td>
</tr>
</tbody>
</table>
Cache Replacement Policies

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - Random
  - FIFO (first-in first-out)
    - When is this a good idea?
  - LRU (least recently used)
    - Fits with temporal locality, LRU = least likely to be used in future
  - NMRU (not most recently used)
    - An easier-to-implement approximation of LRU
    - NMRU=LRU for 2-way set-associative caches
  - Belady’s: replace block that will be used furthest in future
    - Unachievable optimum (but good for comparisons)
  - Which policy is simulated in previous slide?
NMRU and Miss Handling

- Add MRU field to each set
  - MRU data is encoded “way”
  - Hit? update MRU
  - Fill? write enable ~MRU
Physical Cache Layout

- Logical layout
  - Data and tags mixed together

- Physical layout
  - Data and tags in separate RAMs
  - Often multiple sets per line
    - As square as possible
    - Not shown here

hit/miss

address

CPU data

1:0

<<

[10:2]

[31:11]

==

==

0 512
1 513
2 514
510 1022
511 1023
Full-Associativity

- How to implement full (or at least high) associativity?
  - Doing it this way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?