Caches (cont’d)

CS 333
Fall 2006

Reading Assignment and Sample Problems
• Read Chapter 7, Section 7.5
• Chapter 7, problems 7.19, 7.20, 7.21
  – These may show up on some upcoming quiz

Topics
• Hit and Miss ratios
• Cache mapping functions (+ pros, cons)
  – Direct-mapped
  – (Fully) Associative-mapped
  – Block-Set Associative (n-way Set Associative)
• Cache Operations
  – Placement strategy
  – Replacement strategy
  – Read and write policy

Review of Terminology
• Block
  – aka line, minimum unit of information
• Hit
  – data is found in upper level
  – Hit rate – fraction of memory accesses that are found in upper level
  – Hit time – time to access the upper level
  • time to determine if it’s a hit + time to access data
• Miss
  – Miss rate – (1 – hit rate) fraction not found in upper level
  – Miss penalty – Time to replace a block in upper level with corresponding block in lower level + time to deliver block to the processor

Calculating Average Access Time
\[ t_a = (h \cdot t_h) + (m \cdot t_m) \]
- \( t_a \): average access time
- \( h \): hit rate, \( m \): miss rate
- \( t_h \): hit time, \( t_m \): miss time

Hit rate = number of hits / total number of accesses
Miss rate = 1 – hit rate

Speedup = \( \frac{\text{Time without cache}}{\text{Time with cache}} \)

Example
• A direct mapped cache has an access time of 2 ns and takes 40 ns to service a cache miss.
• Without the cache, main memory access time was 50 ns.
• Running benchmarks on designs both with and without caches shows 80% speedup. What is the approximate hit ratio of the cache?
Direct-Mapped Caches

Cache Operations

- Placement strategy
  - Where to place an incoming block in the cache
- Replacement strategy
  - Which block to replace on cache miss
- Read and write policies
  - How to handle reads and writes on cache hits and misses

Cache Mapping Function

Has to be implemented in hardware. Why?

Direct-mapped Cache

- Cache: 256 blocks
- Block: 8 bytes
- Main memory: 64 KB
- 16-bit main memory address

Direct-mapped Cache Operation

Direct-Mapped Example

- Cache – 8KB
  - Block size: 32 bytes per line
  - Direct-mapped
- Main memory – 4 GB (2^{32} bytes ... 32-bit address)
  - Byte-addressable

How many bits are in the Tag, Index, and Offset fields of the 32-bit address?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
What is the Number of Bits to Implement?

- Num\_entries \* (Tag bits + Valid Bit + Data bits)

\[ 2^8 \times (19 + 1 + 256) \]
\[ = 70,656 \]
\[ \approx 8.625\text{KB} \]

Direct-Mapped Caches

- Pros
  - Less hardware
  - Simple block replacement

- Cons
  - Restrictive in block placement
  - Frequently referenced blocks that map to same index will "thrash" (lower performance)

(Fully) Associative Mapped Caches

- Any block from main memory can map to any location in the cache

Fully Associative Example

- Cache: 256 blocks
- Block: 8 bytes
- Main memory: 64 KB
- 16-bit main memory address

Fully Associative Cache Operation

- All locations searched simultaneously

- Cache: 256 blocks
- Block: 8 bytes
- Main memory: 64 KB
- 16-bit main memory address
Fully Associative Example

- Cache – 8KB
  - Block size: 32 bytes per line
  - Fully-associative
- Main memory – 4 GB (2^{32} bytes ... 32-bit address)
  - Byte-addressable

How many bits are in the Tag, Index, and Offset fields of the 32-bit address?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

What is the Number of Bits Required to Implement?

- Num_entries * (Tag bits + Match bit + Valid Bit + Data bits)

2^8 entries * (27 + 1 + 1 + 256 bits)

72,960 bits

≈ 8.91 KB

Compare to Direct-mapped: 8.625 KB ... 2304 more bits

Fully Associative Caches

- Pros
  - Flexible placement
  - Any main memory block can go anywhere in the cache
- Cons
  - Large tag memory
  - Need to search entire tag memory... lots of hardware
  - What's a good replacement policy? (more later)

Block-Set Associative Caches

- Shares properties with direct-mapped and fully associative
- More than one block can occupy the same index location in the cache

Example: 2-way Set Associative

Cache: 256 blocks
Block: 8 bytes
Main memory: 64 KB
16-bit main memory address
2-way Set Associative Example
• Cache – 8KB
  – Block size: 32 bytes per line
  – 2-way set associative
• Main memory – 4 GB (2^{32} bytes … 32-bit address)
  – Byte-addressable

How many bits are in the Tag, Index, and Offset fields of the 32-bit address?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

How Many Bits to Implement?
• Num\_entries * ((num\_ways*tag bits) + (num\_ways * valid bits) + (num\_ways * match bit) + (num\_ways*data bits))

\[
2^7 \times ((2 \times 20) + (2 \times 1) + (2 \times 1) + (2 \times 256))
\]

71,168 bits
≈ 8.68 KB

4-way Set Associative Example
• Cache – 8KB
  – Block size: 32 bytes per line
  – 4-way set associative
• Main memory – 4 GB (2^{32} bytes … 32-bit address)
  – Byte-addressable

How many bits are in the Tag, Index, and Offset fields of the 32-bit address?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

How Many Bits to Implement?
• Num\_entries * ((num\_ways*tag bits) + (num\_ways * valid bits) + (num\_ways * match bit) + (num\_ways*data bits))

\[
2^6 \times ((4 \times 21) + (4 \times 1) + (4 \times 1) + (4 \times 256))
\]

71,424 bits
≈ 8.72 KB

Summary
• For same amount of data storage
  – Direct-mapped
    • smaller tags, smaller implementation area than associative counterparts
  – Fully-associative
    • larger tags, larger total implementation area than direct-mapped counterpart

Read and Write Policies
Write Policies on Cache Hit

- **Write-through**
  - Update both cache and main memory on a write
  - Advantageous when:
    - Few writes
  - Disadvantageous:
    - Many writes (takes time)
- **Write-back**
  - Write to cache only. Update main memory when block is replaced (needs dirty bit per cache block)

Read Miss Policies

- **Read block in from main memory.**
  - Two approaches:
    - Forward desired word first (faster, but more hardware needed)
    - Delay until entire block has been moved to cache

Write Miss Policies

- **Write allocate**
  - Bring block to cache from memory, then update
- **Write-no allocate**
  - Only update main memory (don’t bring into cache)

Block Replacement Strategies

Block Replacement Policies

- If the cache is full
  - Least recently used (LRU)
    - Uses counters
  - Random replacement

Write through caches usually use write-no allocate
Write-back caches usually use write allocate