Lecture 14:
Hardware Caches

- Cache Implementations:
  - Direct-Mapped
  - Fully-Associative
  - Set-Associative
  - Replacement & Write Policies

Notes:
- Lab 5 due today
- Quiz 2, Thu Nov 2

http://xkcd.com/908/
Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else...

<table>
<thead>
<tr>
<th>Access time</th>
<th>Capacity</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cycle</td>
<td>1 KB</td>
<td>Software/Compiler</td>
</tr>
<tr>
<td>2-4 cycles</td>
<td>32 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>10 cycles</td>
<td>256 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>40 cycles</td>
<td>10 MB</td>
<td>Hardware</td>
</tr>
<tr>
<td>200 cycles</td>
<td>10 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>10-100us</td>
<td>100 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>10ms</td>
<td>1 TB</td>
<td>Software/OS</td>
</tr>
</tbody>
</table>

TODAY: Hardware Caches

Later: Software Caches (Virtual Memory)

HW vs SW caches:
- Same objective: fake large, fast, cheap mem
- Conceptually similar
- Different implementations (very different tradeoffs!)
Reminder: Cache Access

- Processor sends address to cache
- Two options:
  - **Cache hit**: Data for this address in cache, returned quickly
  - **Cache miss**: Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)
  - Processor must deal with variable memory access time

![Diagram of cache access](attachment://cache_access_diagram.png)

```
LD 0x6004
LD 0x6034
```

```
Processor: 0x6004 -> Cache: DATA
Processor: 0x6034 -> Cache: DATA
Cache: 0x6034 -> Main Memory: DATA
```

6.004 Computation Structures  L14: Hardware Caches, Slide #3
Basic Cache Algorithm

ON REFERENCE TO Mem[X]:
Look for X among cache tags...

HIT: \( X = TAG(i), \text{ for some cache line } i \)
- READ: return DATA(i)
- WRITE: change DATA(i); Start Write to Mem(X)

MISS: \( X \) not found in TAG of any cache line
- REPLACEMENT SELECTION:
  Select some line \( k \) to hold Mem[X] (Allocation)
- READ: Read Mem[X]
  Set TAG(k)=X, DATA(k)=Mem[X]
- WRITE: Start Write to Mem(X)
  Set TAG(k)=X, DATA(k)= new Mem[X]

Q: How do we “search” the cache?
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^W$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT

**Example: 8-location DM cache ($W=3$)**

32-bit BYTE address

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (27 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.004 Computation Structures
Example: Direct-Mapped Caches

64-line direct-mapped cache $\rightarrow$ 64 indexes $\rightarrow$ 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

TAG: 0x40
INDEX: 0x3
OFFSET: 0x0

HIT, DATA 0x42424242

Would 0x4008 hit?
INDEX: 0x2 $\rightarrow$ tag mismatch $\rightarrow$ miss

What are the addresses of data in indexes 0, 1, and 2?
TAG: 0x58 $\rightarrow$ 0101 1000 iii i00 (substitute line # for iiiii) $\rightarrow$ 0x5800, 0x5804, 0x5808

Part of the address (index bits) is **encoded in the location!**
Tag + Index bits unambiguously identify the data’s address
Block Size

Take advantage of locality: increase block size
- Another advantage: Reduces size of tag memory!
- Potential disadvantage: Fewer blocks in the cache

Example: 4-block, 16-word DM cache

32-bit BYTE address
Tag bits: 26 (=32-4-2)

Valid bit Tag (26 bits) Data (4 words, 16 bytes)

Index bits: 2 (4 indexes)
Block offset bits: 4 (16 bytes/block)
Block Size Tradeoffs

• Larger block sizes...
  – Take advantage of spatial locality
  – Incur larger miss penalty since it takes longer to transfer the block into the cache
  – Can increase the average hit time and miss rate

• Average Access Time (AMAT) = HitTime + MissPenalty*MR

---

**Graphs:**
- **Miss Penalty vs. Block Size**: Increasing block size increases miss penalty.
- **Miss Ratio vs. Block Size**: Smaller block sizes exploit spatial locality, while larger ones compromise locality.
- **AMAT vs. Block Size**: Optimal block size around 64 bytes minimizes AMAT.
Direct-Mapped Cache Problem: Conflict Misses

<table>
<thead>
<tr>
<th>Loop A: Pgm at 1024, data at 37:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Address</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>1025</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Assume:
- 1024-line DM cache
- Block size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing

Inflexible mapping (each address can only be in one cache location) → Conflict misses!

<table>
<thead>
<tr>
<th>Loop B: Pgm at 1024, data at 2048:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Address</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
</tr>
<tr>
<td>1025</td>
</tr>
<tr>
<td>2049</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>2050</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Fully-Associative Cache

Opposite extreme: Any address can be in any location
- No cache index!
- **Flexible** (no conflict misses)
- Expensive: Must compare tags of all entries in parallel to find matching one (can do this in hardware, this is called a CAM)

```
32-bit BYTE address

Tag bits  Offset bits

Tag  Valid bit  Data

0  1  2  3
```
N-way Set-Associative Cache

• Compromise between direct-mapped and fully associative
  – Nomenclature:
    • # Rows = # Sets
    • # Columns = # Ways
    • Set size = #ways
      = “set associativity”
      (e.g., 4-way → 4 entries/set)
    – compare all tags from all ways in parallel

• An N-way cache can be seen as:
  – N direct-mapped caches in parallel

• Direct-mapped and fully-associative are just special cases of N-way set-associative
N-way Set-Associative Cache

Example: 3-way 8-set cache
“Let me count the ways.”

*Elizabeth Barrett Browning*

Potential cache line conflicts during interval $\Delta t$
Associativity Tradeoffs

- More ways...
  - Reduce conflict misses
  - Increase hit time

\[
AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty}
\]

Little additional benefits beyond 4 to 8 ways

[H&P: Fig 5.9]
Associativity Implies Choices

**Issue: Replacement Policy**

**Direct-mapped**
- Compare addr with only one tag
- Location A can be stored in exactly one cache line

**N-way set-associative**
- Compare addr with N tags simultaneously
- Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

**Fully associative**
- Compare addr with each tag simultaneously
- Location A can be stored in any cache line
Replacement Policies

• Optimal policy (Belady’s MIN): Replace the block that is accessed furthest in the future
  – Requires knowing the future...
• Idea: Predict the future from looking at the past
  – If a block has not been used recently, it’s often less likely to be accessed in the near future (a locality argument)
• Least Recently Used (LRU): Replace the block that was accessed furthest in the past
  – Works well in practice
  – Need to keep ordered list of N items \( \rightarrow \) N! orderings
    \( \rightarrow \) \( O(\log_2 N!) = O(N \log_2 N) \) “LRU bits” + complex logic
  – Caches often implement cheaper approximations of LRU
• Other policies:
  – First-In, First-Out (least recently replaced)
  – Random: Choose a candidate at random
    • Not very good, but does not have adversarial access patterns
Write Policy

Write-through: CPU writes are cached, but also written to main memory immediately (stalling the CPU until write is completed). Memory always holds current contents
  – **Simple**, slow, wastes bandwidth

Write-behind: CPU writes are cached; writes to main memory may be buffered. CPU keeps executing while writes are completed in the background
  – **Faster**, still uses lots of bandwidth

Write-back: CPU writes are cached, but not written to main memory until we replace the block. Memory contents can be “stale”
  – **Fastest**, low bandwidth, more complex
  – Commonly implemented in current systems
Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: $TAG(X) == Tag[i]$, for some cache block $i$

- READ: return $Data[i]$
- WRITE: change $Data[i]$; Start Write to Mem[X]

MISS: $TAG(X)$ not found in tag of any cache block that $X$ can map to

- REPLACEMENT SELECTION:
  - Select some line $k$ to hold Mem[X]
  - Write Back: Write $Data[k]$ to Mem[Address from Tag[k]]

- READ: Read Mem[X]
  - Set $Tag[k] = TAG(X)$, $Data[k] = Mem[X]$

- WRITE: Start Write to Mem[X]
  - Set $Tag[k] = TAG(X)$, $Data[k] = \text{new Mem[X]}$
Write-Back with “Dirty” Bits

Add 1 bit per block to record whether block has been written to. Only write back dirty blocks.

ON REFERENCE TO Mem[X]: Look for TAG(X) among tags...

HIT: TAG(X) == Tag[i], for some cache block i
  • READ: return Data[i]
  • WRITE: change Data[i] Start Write to Mem[X] D[i]=1

MISS: TAG(X) not found in tag of any cache block that X can map to
  • REPLACEMENT SELECTION:
    ▪ Select some block k to hold Mem[X]
    ▪ If D[k] == 1 (Writeback) Write Data[k] to Mem[Address of Tag[k]]
  • READ: Read Mem[X]; Set Tag[k] = TAG(X), Data[k] = Mem[X], D[k]=0
  • WRITE: Start Write to Mem[X] D[k]=1
    ➢ Set Tag[k] = TAG(X), Data[k] = new Mem[X]
Summary: Cache Tradeoffs

\[ AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty} \]

- Larger cache size: Lower miss rate, higher hit time
- Larger block size: Trade off spatial for temporal locality, higher miss penalty
- More associativity (ways): Lower miss rate, higher hit time
- More intelligent replacement: Lower miss rate, higher cost
- Write policy: Lower bandwidth, more complexity
- How to navigate all these dimensions? Simulate different cache organizations on real programs
Example: Comparing Hit Rates

- 3 Caches: DM, 2-Way, FA: each has 8 (4 byte) words
Example: Comparing Hit Rates

Access following addresses repeatedly: 0, 16, 4, 36, ...

<table>
<thead>
<tr>
<th>DM</th>
<th>2-Way</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 36</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>36</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36</td>
</tr>
</tbody>
</table>

16 = 0b010000  
DM index = 100  
2-Way index = 00

4 = 0b000100  
DM index = 001  
2-Way index = 01

36 = 0b100100  
DM index = 001  
2-Way index = 01
Example: Comparing Hit Rates

Access following addresses repeatedly: 0, 16, 4, 36, ...

DM: 50% hit rate
2-Way: 100% hit rate
FA: 100% hit rate
**Example 2: Comparing Hit Rates**

Access: 0, 4, 8, 12, 16, 20, 24, 28, 32, ...

<table>
<thead>
<tr>
<th>DM</th>
<th>2-Way</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 32</td>
<td>0, 32, 16</td>
<td>0, 32</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>24</td>
<td>28</td>
<td>8</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

DM: Hit rate = $\frac{7}{9}$   2-Way: Hit rate = $\frac{6}{9}$   FA: Hit rate = 0%
Example 3: Comparing Hit Rates

- Access: 0, 4, 8, 12, 32, 36, 40, 44, 16, ...

DM

| 0, 32 |
| 4, 36 |
| 8, 40 |
| 12, 44 |
| 16 |

2-Way

| 0, 16, 32 |
| 4 |
| 8 |
| 12 |

| 32, 0, 16 |
| 36 |
| 40 |
| 44 |

FA

| 0, 16 |
| 4, 0 |
| 8, 4 |
| 12, 8 |
| 32, 12 |
| 36, 32 |
| 40, 36 |
| 44, 40 |

DM: Hit rate = 1/9
2-Way: Hit rate = 6/9
FA: Hit rate = 0%