CS 61C: Great Ideas in Computer Architecture

The Memory Hierarchy, Fully Associative Caches

Instructor: Alan Christopher
Review of Last Lecture

• Floating point (single and double precision) approximates real numbers
  – Exponent field uses biased notation
  – Special cases: 0, ±∞, NaN, denorm
  – High precision for small numbers
  – Low precision for large numbers

• Performance measured in latency or bandwidth

• Latency measurement:
  – CPU Time = Instructions × CPI × Clock Cycle Time
  – Affected by different components of the computer
Great Idea #3: Principle of Locality / Memory Hierarchy
Agenda

• Memory Hierarchy Overview
• Administrivia
• Fully Associative Caches
• Cache Reads and Writes
Storage in a Computer

• Processor
  - Holds data in register files (~ 100 B)
  - Registers accessed on sub-nanosecond timescale

• Memory ("main memory")
  - More capacity than registers (~ GiB)
  - Access time ~ 50-100 ns

• Hundreds of clock cycles per memory access?!
1989 first Intel CPU with cache on chip
1998 Pentium III has two cache levels on chip

“Moore’s Law”

Processor-Memory Gap

μProc 55%/year
(2X/1.5yr)

DRAM 7%/year
(2X/10yrs)

Processor-Memory Performance Gap
(grows 50%/year)
Library Analogy

• Writing a report on a specific topic
  – e.g. history of the computer (your internet is out)

• While at library, take books from shelves and keep them on desk

• If need more, go get them and bring back to desk
  – Don’t return earlier books since might still need them
  – Limited space on desk; which books do we keep?

• You hope these ~10 books on desk enough to write report
  – Only 0.00001% of the books in UC Berkeley libraries!
Principle of Locality (1/3)

- *Principle of Locality:* Programs access only a small portion of the full address space at any instant of time
  - **Recall:** Address space holds both code and data
  - Loops and sequential instruction execution mean generally localized code access
  - Stack and Heap try to keep your data together
  - Arrays and structs naturally group data you would access together
Principle of Locality (2/3)

• **Temporal Locality** (locality in time)
  - Go back to the same book on desk multiple times
  - If a memory location is referenced then it will tend to be referenced again soon

• **Spatial Locality** (locality in space)
  - When go to shelves, grab many books on computers since related books are stored together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
Principle of Locality (3/3)

• We exploit the principle of locality in hardware via a \textit{memory hierarchy} where:
  – Levels closer to processor are faster (and more expensive per bit so smaller)
  – Levels farther from processor are larger (and less expensive per bit so slower)

• \textbf{Goal:} Create the illusion of memory being almost as fast as fastest memory and as large as biggest memory of the hierarchy
Memory Hierarchy Schematic

- Processor
- Level 1
- Level 2
- Level 3
- \( \ldots \)
- Level \( n \)

Smaller, Faster, More expensive
Bigger, Slower, Cheaper
Cache Concept

• Introduce intermediate hierarchy level: memory cache, which holds a copy of a subset of main memory
  - As a pun, often use $ (“cash”) to abbreviate cache (e.g. D$ = Data Cache, L1$ = Level 1 Cache)

• Modern processors have separate caches for instructions and data, as well as several levels of caches implemented in different sizes

• Implemented with same IC processing technology as CPU and integrated on-chip – faster but more expensive than main memory
Memory Hierarchy Technologies

• Caches use static RAM (SRAM)
  - Fast (typical access times of 0.5 to 2.5 ns)
  - Low density (6 transistor cells), higher power, expensive ($2000 to $4000 per GB in 2011)
  - *Static:* content will last as long as power is on

• Main memory uses dynamic RAM (DRAM)
  - High density (1 transistor cells), lower power, cheaper ($20 to $40 per GB in 2011)
  - Slower (typical access times of 50 to 70 ns)
  - *Dynamic:* needs to be “refreshed” regularly (~ every 8 ms)
Inclusive: data in L1$ ⊆ data in L2$ ⊆ data in MM ⊆ data in SM

Block: Unit of transfer between memory and cache

Processor

4-8 bytes (word)

L1$

8-32 bytes (block)

L2$

16-128 bytes (block)

Main Memory

4,096+ bytes (page)

Secondary Memory
The Cache Block

• The *Cache Block* is the fundamental unit of memory that caches work with
  – Always retrieve one cache block when grabbing values from lower levels
  – Store data together in blocks
  – Evict blocks at a time (when necessary)
• Only access data at finer granularity when serving the level above
Managing the Hierarchy

• registers ↔ memory
  - By compiler (or assembly level programmer)

• cache ↔ main memory
  - By the cache controller hardware

• main memory ↔ disks (secondary storage)
  - By the OS (virtual memory, which is a later topic)
  - Virtual to physical address mapping assisted by the hardware (TLB)
  - By the programmer (files)
Typical Memory Hierarchy

On-Chip Components

- Control
- Datapath
  - RegFile
- Second Level Cache (SRAM)
- Main Memory (DRAM)
- Secondary Memory (Disk or Flash)

Cost/bit:
- highest
- lowest

Speed:
- ½’s
- 1’s
- 10’s
- 100’s
- 1,000,000’s

Size:
- 100’s
- 10K’s
- M’s
- G’s
- T’s

( pairwise comparisons for size)
Review So Far

- **Goal:** present the programmer with as much memory as the *largest* memory at ≈ the speed of the *fastest* memory
- **Approach:** Memory Hierarchy
  - Successively higher levels contain “most used” data from lower levels
  - Exploits *temporal and spatial locality*
  - Caches fill in the space between CPU and DRAM
Agenda

• Memory Hierarchy Overview
• Administrivia
• Fully Associative Caches
• Cache Reads and Writes
Administrivia

• Project 1 check-in (T hours so far)
  - (blue) 0 <= T < 6
  - (green) 6 <= T < 12
  - (purple) 12 <= T < 18
  - (yellow) 18 <= T
Administrivia

• Project 1 check-in – which parts have you finished?
  – Test suite?
  – Lexer?
  – Parser?
  – Code Generator?
  – Debugging?
Agenda

- Memory Hierarchy Overview
- Administrivia
- Fully Associative Caches
- Cache Reads and Writes
Cache Management

• What is the overall organization of blocks we impose on our cache?
  – Where do we put a block of data from memory?
  – How do we know if a block is already in cache?
  – How do we quickly find a block when we need it?
  – When do we replace something in the cache?
General Notes on Caches (1/4)

- **Recall:** Memory is *byte-addressed*
- We haven’t specified the size of our “blocks,” but will be multiple of word size (32-bits)
  - How do we access individual words or bytes within a block?
- Cache is smaller than memory
  - Can’t fit all blocks at once, so multiple blocks in memory must map to the same slot in cache
  - Need some way of identifying which memory block is currently in each cache slot
General Notes on Caches (2/4)

• **Recall:** hold subset of memory in a place that’s faster to access
  – Return data to you when you request it
  – If cache doesn’t have it, then fetches it for you

• Cache must be able to check/identify its current contents

• What does cache initially hold?
  – Garbage! Cache considered “cold”
  – Keep track with Valid bit
General Notes on Caches (3/4)

- Effect of block size (K Bytes):
  - Spatial locality dictates our blocks consist of adjacent bytes, which differ in address by 1
  - Offset field: Lowest bits of memory address can be used to index to specific bytes within a block
    - Block size needs to be a power of two (in bytes)
    - (address) modulo (# of bytes in a block)
General Notes on Caches (4/4)

- Effect of cache size (C Bytes):
  - “Cache Size” refers to total stored data
  - Determines number of blocks the cache can hold (C/K blocks)
  - Tag field: Leftover upper bits of memory address determine which portion of memory the block came from (identifier)
Fully Associative Caches

• Each memory block can map anywhere in the cache (fully associative)
  – Most efficient use of space
  – Least efficient to check

• To check a fully associative cache:
  – Look at ALL cache slots in parallel
  – If Valid bit is 0, then ignore
  – If Valid bit is 1 and Tag matches, then return that data
Block Replacement Policies

• Which block do you replace?
  – Use a *cache block replacement policy*
  – There are many (most are intuitively named), but we will just cover a few in this class
    http://en.wikipedia.org/wiki/Cache_algorithms#Examples

• Of note:
  – Random Replacement
  – Least Recently Used (LRU): requires some “management bits”
Fully Associative Cache Address Breakdown

• Memory address fields:

- Tag
- Offset

31
0

T bits
O bits

• Meaning of the field sizes:
  - O bits $\leftrightarrow 2^O$ bytes/block $= 2^{O-2}$ words/block
  - T bits = A - O, where A = # of address bits (A = 32 here)
Caching Terminology (1/2)

• When reading memory, 3 things can happen:
  - **Cache hit:**
    Cache holds a valid copy of the block, so return the desired data
  - **Cache miss:**
    Cache does not have desired block, so fetch from memory and put in empty (invalid) slot
  - **Cache miss with block replacement:**
    Cache does not have desired block and is full, so discard one and replace it with desired data
Caching Terminology (2/2)

• How effective is your cache?
  - Want to max cache hits and min cache misses
  - **Hit rate (HR):** Percentage of memory accesses in a program or set of instructions that result in a cache hit
  - **Miss rate (MR):** Like hit rate, but for cache misses
    \[ MR = 1 - HR \]

• How fast is your cache?
  - **Hit time (HT):** Time to access cache (including Tag comparison)
  - **Miss penalty (MP):** Time to replace a block in the cache from a lower level in the memory hierarchy
Fully Associative Cache Implementation

• What’s actually in the cache?
  – Each cache slot contains the actual data block
    \(8 \times K = 8 \times 2^0\) bits
  – Tag field of address as identifier \((T\) bits)
  – Valid bit \((1\) bit)
  – Any necessary replacement management bits
    (“LRU bits” – variable \# of bits)

• Total bits in cache
  \[= \# \text{ slots} \times (8\times K + T + 1) + ?\]
  \[= \left(\frac{C}{K}\right) \times (8\times 2^0 + T + 1) + ? \text{ bits}\]
FA Cache Examples (1/4)

• Cache parameters:
  - Fully associative, address space of 64B, block size of 1 word, cache size of 4 words, LRU (5 bits)

• Address Breakdown:
  - 1 word = 4 bytes, so $O = \log_2(4) = 2$
  - $A = \log_2(64) = 6$ bits, so $T = 6 - 2 = 4$

• Bits in cache
  \[
  = (4/1) \times (8 \times 2^2 + 4 + 1) + 5 = 153 \text{ bits}
  \]
FA Cache Examples (1/4)

- Cache parameters:
  - Fully associative, address space of 64B, block size of 1 word, cache size of 4 words, LRU (5 bits)
  - Offset – 2 bits, Tag – 4 bits

- 37 bits per slot, 153 bits to implement with LRU
FA Cache Examples (2/4)

1) Consider the sequence of memory address accesses

Starting with a cold cache:

\[0 \quad 2 \quad 4 \quad 8 \quad 20 \quad 16 \quad 0 \quad 2\]

**0 miss**

\[
\begin{array}{cccc}
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
\end{array}
\]

**2 hit**

\[
\begin{array}{cccc}
1 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
\end{array}
\]

**4 miss**

\[
\begin{array}{cccc}
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
\end{array}
\]

**8 miss**

\[
\begin{array}{cccc}
0 & 0000 & 0x?? & 0x?? & 0x?? & 0x?? \\
\end{array}
\]
FA Cache Examples (2/4)

1) Consider the sequence of memory address accesses

Starting with a cold cache:

<table>
<thead>
<tr>
<th>Accesses</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
</tr>
<tr>
<td>20</td>
<td>M</td>
</tr>
<tr>
<td>16</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
</tr>
</tbody>
</table>

0  miss

<table>
<thead>
<tr>
<th>Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>M[0]</td>
</tr>
<tr>
<td>10001</td>
<td>M[1]</td>
</tr>
<tr>
<td>100010</td>
<td>M[2]</td>
</tr>
<tr>
<td>10010</td>
<td>M[3]</td>
</tr>
<tr>
<td>10101</td>
<td>M[4]</td>
</tr>
</tbody>
</table>

16 miss

<table>
<thead>
<tr>
<th>Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>M[0]</td>
</tr>
<tr>
<td>10001</td>
<td>M[1]</td>
</tr>
<tr>
<td>100010</td>
<td>M[2]</td>
</tr>
<tr>
<td>10010</td>
<td>M[3]</td>
</tr>
<tr>
<td>10101</td>
<td>M[4]</td>
</tr>
</tbody>
</table>

2  hit

<table>
<thead>
<tr>
<th>Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>M[16]</td>
</tr>
<tr>
<td>10001</td>
<td>M[17]</td>
</tr>
<tr>
<td>100010</td>
<td>M[18]</td>
</tr>
<tr>
<td>10010</td>
<td>M[19]</td>
</tr>
<tr>
<td>10101</td>
<td>M[20]</td>
</tr>
</tbody>
</table>

• 8 requests, 6 misses – HR of 25%
FA Cache Examples (3/4)

2) Same requests, but reordered
Starting with a cold cache: 0 2 2 0 16 20 8 4

0 miss
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
2 hit
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
2 hit
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
0 hit
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
0000 0x?? 0x?? 0x?? 0x??
2) Same requests, but reordered

Starting with a cold cache:

```
16 miss
0   2   2   0   16   20   8   4
M    H    H    H    H
```

```
20 miss
0   2   2   0   16   20   8   4
M    H    H    H    H
```

```
8 miss
0   2   2   0   16   20   8   4
M    H    H    H    H
```

```
4 miss
0   2   2   0   16   20   8   4
M    H    H    H    H
```

- 8 requests, 5 misses – ordering matters!
3) Original sequence, but double block size

Starting with a cold cache:  0  2  4  8  20  16  0  2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>miss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
<td>0x??</td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|---|------|------|------|------|------|------|------|------|------|
3) Original sequence, but double block size

Starting with a cold cache:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>miss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 8 requests, 4 misses – cache parameters matter!
**Question:**
Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

LRU  
10

\[
\begin{array}{cccccc}
(A) & 0 & 2 & 12 & 4 & 16 & 8 & 0 & 6 \\
(B) & 0 & 8 & 4 & 16 & 0 & 12 & 6 & 2 \\
(C) & 6 & 12 & 4 & 8 & 2 & 16 & 0 & 0 \\
(D) & 2 & 8 & 0 & 4 & 6 & 16 & 12 & 0 \\
\end{array}
\]
Technology Break
Agenda

- Memory Hierarchy Overview
- Administrivia
- Fully Associative Caches
- Cache Reads and Writes
Memory Accesses

• The picture so far:

  ![Diagram]

  - Cache is separate from memory
    - Possible to hold different data?
Cache Reads and Writes

• Want to handle reads and writes quickly while maintaining consistency between cache and memory (i.e. both know about all updates)

• Here we assume the use of separate instruction and data caches (I$ and D$)
  – Read from both
  – Write only to D$ (assume no self-modifying code)
Handling Cache Hits

• Read hits (I$ and D$)
  – Fastest possible scenario, so want more of these

• Write hits (D$)
  – Write-Through Policy: Always write data to cache and to memory (*through* cache)
    • Forces cache and memory to always be consistent
    • Slow! (every memory access is long)
    • Include a *Write Buffer* that updates memory in parallel with processor
Handling Cache Hits

• Read hits (I$ and D$)
  – Fastest possible scenario, so want more of these

• Write hits (D$)
  – **Write-Back Policy:** Write data **only to cache**, then update memory when block is removed
    • Allows cache and memory to be inconsistent
    • Multiple writes collected in cache; single write to memory per block
  
  • **Dirty bit:** Extra bit per cache row that is set if block was written to (is “dirty”) and needs to be written back
Handling Cache Misses

• Read misses (I$ and D$)
  - Stall execution, fetch block from memory, put in cache, send requested data to processor, resume

• Write misses (D$)
  - Write allocate: Fetch block from memory, put in cache, execute a write hit
    • Works with either write-through or write-back
    • Ensures cache is up-to-date after write miss
Handling Cache Misses

• Read misses (I$ and D$)
  – Stall execution, fetch block from memory, put in cache, send requested data to processor, resume

• Write misses (D$)
  – No-write allocate: Skip cache altogether and write directly to memory
    • Cache is never up-to-date after write miss
    • Ensures memory is always up-to-date
Updated Cache Picture

• Fully associative, write through
  – Same as previously shown
• Fully associative, write back

<table>
<thead>
<tr>
<th>Slot</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>xxxx</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>xxxx</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td>xxxx</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>xxxx</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
<td>0??</td>
</tr>
</tbody>
</table>

• Write allocate/no-write allocate
  – Affects behavior, not design
Summary (1/2)

• Memory hierarchy exploits principle of locality to deliver lots of memory at fast speeds
• Fully Associative Cache: Every block in memory maps to any cache slot
  – Offset to determine which byte within block
  – Tag to identify if it’s the block you want
• Replacement policies: random and LRU
• Cache params: block size (K), cache size (C)
Summary (2/2)

• Cache read and write policies:
  – *Write-back* and *write-through* for hits
  – *Write allocate* and *no-write allocate* for misses
  – Cache needs *dirty* bit if write-back