CS 61C: Great Ideas in Computer Architecture (Machine Structures) Caches Part 2

Instructors:
John Wawrzynek & Vladimir Stojanovic
http://inst.eecs.berkeley.edu/~cs61c/
Typical Memory Hierarchy

- **Principle of locality + memory hierarchy** presents programmer with ≈ as much memory as is available in the **cheapest** technology at the ≈ speed offered by the **fastest** technology
Review: Adding Cache to Computer
Memory Block-addressing example

<table>
<thead>
<tr>
<th>address</th>
<th>8-byte Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
</tr>
<tr>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

2 LSBs are 0

3 LSBs are 0

Byte offset in block

10/20/15
Block number aliasing example

12-bit memory addresses, 16 Byte blocks

<table>
<thead>
<tr>
<th>Block #</th>
<th>Block # mod 8</th>
<th>Block # mod 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>82 010100100000</td>
<td>2 010100100000</td>
<td>0 010100100000</td>
</tr>
<tr>
<td>83 010100110000</td>
<td>3 010100110000</td>
<td>1 010100110000</td>
</tr>
<tr>
<td>84 010101000000</td>
<td>4 010101000000</td>
<td>0 010101000000</td>
</tr>
<tr>
<td>85 010101010000</td>
<td>5 010101010000</td>
<td>1 010101010000</td>
</tr>
<tr>
<td>86 010101100000</td>
<td>6 010101100000</td>
<td>0 010101100000</td>
</tr>
<tr>
<td>87 010101110000</td>
<td>7 010101110000</td>
<td>1 010101110000</td>
</tr>
<tr>
<td>88 010110000000</td>
<td>0 010110000000</td>
<td>0 010110000000</td>
</tr>
<tr>
<td>89 010110010000</td>
<td>1 010110010000</td>
<td>1 010110010000</td>
</tr>
<tr>
<td>90 010110100000</td>
<td>2 010110100000</td>
<td>0 010110100000</td>
</tr>
<tr>
<td>91 010110110000</td>
<td>3 010110110000</td>
<td>1 010110110000</td>
</tr>
</tbody>
</table>

10/20/15
Caches Review

- Principle of Locality
  - Temporal Locality and Spatial Locality
- Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
- Cache – copy of data in lower level of memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit
- Cache design organization choices:
  - Fully Associative, Set-Associative, Direct-Mapped
Cache Organizations

• “Fully Associative”: Block can go anywhere
  – First design in lecture
  – Note: No Index field, but 1 comparator/block

• “Direct Mapped”: Block goes one place
  – Note: Only 1 comparator
  – Number of sets = number blocks

• “N-way Set Associative”: N places for a block
  – Number of sets = number of blocks / N
  – N comparators
  – *Fully Associative*: $N = \text{number of blocks}$
  – *Direct Mapped*: $N = 1$
Processor Address Fields used by Cache Controller

- **Block Offset**: Byte address within block
- **Set Index**: Selects which set
- **Tag**: Remaining portion of processor address

![Processor Address Diagram](image)

- Size of Index = $\log_2$ (number of sets)
- Size of Tag = Address size – Size of Index – $\log_2$ (number of bytes/block)
One word blocks, cache size = 1K words (or 4KB)

- Valid bit ensures something useful in cache for this index
- Compare Tag with upper part of Address to see if a Hit
- Read data from cache instead of memory if a Hit

Diagram:
- Valid bit
- Tag
- Index
- Data
- Byte offset
- Comparator
Four-Way Set-Associative Cache

- $2^8 = 256$ sets each with four ways (each with one block)

![Diagram of a four-way set-associative cache]

- Each set has 256 blocks.
- Each way has one block.
- The byte offset is used to determine the block within each way.
- The set index, tag, and data are used to determine the cache entry.
- A hit occurs if the data matches, and the corresponding block is selected.

Set Index

Tag

Index

8

V Tag Data

Way 0

Way 1

Way 2

Way 3

Hit

Data

4x1 select

32

31 30  ... 13 12 11  ... 2 1 0

Byte offset

0

1

2

253

254

255

0

1

2

253

254

255

0

1

2

253

254

255

0

1

2

253

254

255
Handling Stores with Write-Through

• Store instructions write to memory, changing values
• Need to make sure cache and memory have same values on writes: 2 policies

1) Write-Through Policy: write cache and write *through* the cache to memory
   – Every write eventually gets to memory
   – Too slow, so include Write Buffer to allow processor to continue once data in Buffer
   – Buffer updates memory in parallel to processor
Write-Through Cache

- Write both values in cache and in memory
- Write buffer stops CPU from stalling if memory cannot keep up
- Write buffer may have multiple entries to absorb bursts of writes
- What if store misses in cache?
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  - Called “Dirty” bit (writing makes it “dirty”)
Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - “Write-allocate” policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.
Write-Through vs. Write-Back

• Write-Through:
  – Simpler control logic
  – More predictable timing simplifies processor control logic
  – Easier to make reliable, since memory always has copy of data (big idea: Redundancy!)

• Write-Back
  – More complex control logic
  – More variable timing (0,1,2 memory accesses per cache access)
  – Usually reduces write traffic
  – Harder to make reliable, sometimes cache has only copy of data
Administrivia

• Project 3-1 due date Wednesday 10/21.
• Project 3-2 due date now 10/28 (release 10/21)

• Midterm 1:
  – grades posted
**Cache (Performance) Terms**

- **Hit rate**: fraction of accesses that hit in the cache
- **Miss rate**: $1 - \text{Hit rate}$
- **Miss penalty**: time to replace a block from lower level in memory hierarchy to cache
- **Hit time**: time to access cache memory (including tag comparison)

- Abbreviation: “$” = cache (A Berkeley innovation!)
Average Memory Access Time (AMAT)

• Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses in the cache

AMAT = Time for a hit
+ Miss rate × Miss penalty
Clickers/Peer instruction

AMAT = Time for a hit + Miss rate x Miss penalty

Given a 200 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache hit time of 1 clock cycle, what is AMAT?

☐ A: ≤200 psec

☐ B: 400 psec

☐ C: 600 psec

☐ D: ≥800 psec
Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers:
  
  Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4
Example: Direct-Mapped Cache with 4 Single-Word Blocks, Worst-Case Reference String

- Consider the main memory address reference string of word numbers: 0 4 0 4 0 4 0 4

Start with an empty cache - all blocks initially marked as not valid

- 8 requests, 8 misses
- Ping-pong effect due to conflict misses - two memory locations that map into the same cache block
Alternative Block Placement Schemes

• DM placement: mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
• SA placement: four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
• FA placement: mem block 12 can appear in any cache blocks
Example: 2-Way Set Associative $\$
(4 words = 2 sets x 2 ways per set)

Q: How do we find it?
Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)

Q: Is it there?
Compare all the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache

One word blocks
Two low order bits define the byte in the word (32b words)
Example: 4 Word 2-Way SA $ 
Same Reference String

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

```
0 4 0 4 0 4 0 4
```
Example: 4-Word 2-Way SA $ 
Same Reference String

- Consider the main memory address reference string

<table>
<thead>
<tr>
<th></th>
<th>0 miss</th>
<th>4 miss</th>
<th>0 hit</th>
<th>4 hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Mem(0)</td>
<td>Mem(0)</td>
<td>Mem(0)</td>
<td>Mem(0)</td>
</tr>
<tr>
<td></td>
<td>Mem(0)</td>
<td>Mem(0)</td>
<td>Mem(0)</td>
<td>Mem(0)</td>
</tr>
<tr>
<td>010</td>
<td>Mem(4)</td>
<td>Mem(4)</td>
<td>Mem(4)</td>
<td>Mem(4)</td>
</tr>
</tbody>
</table>

• 8 requests, 2 misses

• Solves the ping-pong effect in a direct-mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!
Four-Way Set-Associative Cache

- \(2^8 = 256\) sets each with four ways (each with one block)

```
Index  V   Tag   Data
0      0    0    Way 0
1      1    1    Way 1
2      2    2    Way 2
3      3    3    Way 3
```

Byte offset

4x1 select

Hit
Total size of $ in blocks is equal to number of sets $ \times $ associativity. For fixed $ size and fixed block size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $ is same as a fully associative $.
Range of Set-Associative Caches

• For a fixed-size cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Word offset</th>
<th>Byte offset</th>
</tr>
</thead>
</table>
Range of Set-Associative Caches

- For a *fixed-size* cache and fixed block size, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Word offset</th>
<th>Byte offset</th>
</tr>
</thead>
</table>

**Decreasing associativity**
- Direct mapped (only one way)
- Smaller tags, only a single comparator

**Increasing associativity**
- Fully associative (only one set)
  - Tag is all the bits except block and byte offset
Total Cache Capacity =

Associativity \times \# \text{ of sets} \times \text{block}_\text{size}

Bytes = \text{blocks/set} \times \text{sets} \times \text{Bytes/block}

C = N \times S \times B

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Byte Offset</th>
</tr>
</thead>
</table>

address\_size = tag\_size + index\_size + offset\_size

= tag\_size + \log_2(S) + \log_2(B)
Clickers/Peer Instruction

- For a cache with constant total capacity, if we increase the number of ways by a factor of 2, which statement is false:
  - A: The number of sets could be doubled
  - B: The tag width could decrease
  - C: The block size could stay the same
  - D: The block size could be halved
  - E: Tag width must increase
Total Cache Capacity =

\[ \text{Associativity} \times \# \text{ of sets} \times \text{block}_\text{size} \]

\[ \text{Bytes} = \text{blocks}/\text{set} \times \text{sets} \times \text{Bytes}/\text{block} \]

\[ C = N \times S \times B \]

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Byte Offset</th>
</tr>
</thead>
</table>

\[ \text{address}_\text{size} = \text{tag}_\text{size} + \text{index}_\text{size} + \text{offset}_\text{size} \]

\[ = \text{tag}_\text{size} + \log_2(S) + \log_2(B) \]

Clicker Question: C remains constant, S and/or B can change such that

\[ C = 2N \times (SB)' \Rightarrow (SB)' = SB/2 \]

\[ \text{Tag}_\text{size} = \text{address}_\text{size} - (\log_2(S) + \log_2(B)) = \text{address}_\text{size} - \log_2(SB) \]

\[ = \text{address}_\text{size} - \log_2(SB/2) \]

\[ = \text{address}_\text{size} - (\log_2(SB) - 1) \]
Costs of Set-Associative Caches

• N-way set-associative cache costs
  – N comparators (delay and area)
  – MUX delay (set selection) before data is available
  – Data available after set selection (and Hit/Miss decision).
    DM $: block is available before the Hit/Miss decision
    • In Set-Associative, not possible to just assume a hit and continue and recover later if it was a miss

• When miss occurs, which way’s block selected for replacement?
  – Least Recently Used (LRU): one that has been unused the longest (principle of temporal locality)
    • Must track when each way’s block was used relative to other blocks in the set
    • For 2-way SA $, one bit per set → set to 1 when a block is referenced; reset the other way’s bit (i.e., “last used”)
Cache Replacement Policies

• Random Replacement
  – Hardware randomly selects a cache evict

• Least-Recently Used
  – Hardware keeps track of access history
  – Replace the entry that has not been used for the longest time
  – For 2-way set-associative cache, need one bit for LRU replacement

• Example of a Simple “Pseudo” LRU Implementation
  – Assume 64 Fully Associative entries
  – Hardware replacement pointer points to one cache entry
  – Whenever access is made to the entry the pointer points to:
    • Move the pointer to the next entry
  – Otherwise: do not move the pointer
  – (example of “not-most-recently used” replacement policy)
Benefits of Set-Associative Caches

- Choice of DM $\rightarrow$ versus SA $\rightarrow$ depends on the cost of a miss versus the cost of implementation.

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate).
Understanding Cache Misses: The 3Cs

• **Compulsory** (cold start or process migration, 1\textsuperscript{st} reference):
  – First access to block impossible to avoid; small effect for long running programs
  – Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

• **Capacity**:
  – Cache cannot contain all blocks accessed by the program
  – Solution: increase cache size (may increase access time)

• **Conflict** (collision):
  – *Multiple memory locations mapped to the same cache location*
  – **Solution 1**: increase cache size
  – **Solution 2**: increase associativity (may increase access time)
How to Calculate 3C’s using Cache Simulator

1. **Compulsory**: set cache size to infinity and fully associative, and count number of misses

2. **Capacity**: Change cache size from infinity, usually in powers of 2, and count misses for each reduction in size
   - 16 MB, 8 MB, 4 MB, ... 128 KB, 64 KB, 16 KB

3. **Conflict**: Change from fully associative to n-way set associative while counting misses
   - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way
• Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  – Compulsory misses 0.006%; not visible
  – Capacity misses, function of cache size
  – Conflict portion depends on associativity and cache size
Improving Cache Performance

AMAT = Time for a hit + Miss rate x Miss penalty

• Reduce the time to hit in the cache
  – E.g., Smaller cache

• Reduce the miss rate
  – E.g., Bigger cache

• Reduce the miss penalty
  – E.g., Use multiple cache levels
Impact of Larger Cache on AMAT?

- 1) Reduces misses (what kind(s)?)
- 2) Longer Access time (Hit time): smaller is faster
  - Increase in hit time will likely add another stage to the pipeline
- At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance
- Computer architects expend considerable effort optimizing organization of cache hierarchy – big impact on performance and power!
Clickers: Impact of longer cache blocks on misses?

• For fixed total cache capacity and associativity, what is effect of longer blocks on each type of miss:
  – A: Decrease, B: Unchanged, C: Increase

• Compulsory?

• Capacity?

• Conflict?
Clickers: Impact of longer blocks on AMAT

• For fixed total cache capacity and associativity, what is effect of longer blocks on each component of AMAT:
  – A: Decrease, B: Unchanged, C: Increase

• Hit Time?

• Miss Rate?

• Miss Penalty?
Clickers/Peer Instruction:
For fixed capacity and fixed block size, how does increasing associativity effect AMAT?

A: Increases hit time, decreases miss rate
B: Decreases hit time, decreases miss rate
C: Increases hit time, increases miss rate
D: Decreases hit time, increases miss rate
Cache Design Space

• Several interacting dimensions
  – Cache size
  – Block size
  – Associativity
  – Replacement policy
  – Write-through vs. write-back
  – Write allocation

• Optimal choice is a compromise
  – Depends on access characteristics
    • Workload
    • Use (I-cache, D-cache)
  – Depends on technology / cost

• Simplicity often wins
And, In Conclusion ... 

• Name of the Game: Reduce AMAT
  – Reduce Hit Time
  – Reduce Miss Rate
  – Reduce Miss Penalty

• Balance cache parameters (Capacity, associativity, block size)