CS 61C: Great Ideas in Computer Architecture
Caches Part 2

Instructors:
Nicholas Weaver & Vladimir Stojanovic
http://inst.eecs.berkeley.edu/~cs61c/fa15
You Are Here!

Software

- Parallel Requests
  Assigned to computer
e.g., Search “Katz”

- Parallel Threads
  Assigned to core
e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
e.g., Add of 4 pairs of words

Hardware

- Harness Parallelism & Achieve High Performance

Hardware descriptions

- All gates @ one time

Programming Languages
What kind of locality are we taking advantage of?
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 (32-bits total)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set Index</th>
<th>Block offset</th>
</tr>
</thead>
</table>

- Size of Index = \( \log_2 \) (number of sets)
- Size of Tag = Address size – Size of Index – \( \log_2 \) (number of bytes/block)
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 with set index, tag, and data fields.]
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 you need to include a 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
- 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
Write Policy Choices

• Cache hit:
  – **write through**: writes both cache & memory on every access
    • Generally higher memory traffic but simpler pipeline & cache design
  – **write back**: writes cache only, memory `written only when dirty entry evicted`
    • A dirty bit per line reduces write-back traffic
    • Must handle 0, 1, or 2 accesses to memory for each load/store

• Cache miss:
  – **no write allocate**: only write to main memory
  – **write allocate** (aka fetch on write): fetch into cache

• Common combinations:
  – write through and no write allocate
  – write back with write allocate
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

\[
AMAT = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}
\]
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: \(0 \ 4 \ 0 \ 4 \ 0 \ 4 \ 0 \ 4\)

Start with an empty cache - all blocks initially marked as not valid.
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: 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 0 4
```

```
0

4

0

4
```
Example: 4-Word 2-Way SA

Same Reference String

- Consider the main memory address reference string

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

```
0  4  0  4  0  4  0  4  4
```

- 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!
Different Organizations of an Eight-Block Cache

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 $.

<table>
<thead>
<tr>
<th>Block</th>
<th>Tag</th>
<th>Data</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>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**One-way set associative**

*(direct mapped)*

**Two-way set associative**

**Four-way set associative**

**Eight-way set associative (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>Decreasing associativity</th>
<th>Increasing associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped (only one way)</td>
<td>Fully associative (only one set)</td>
</tr>
<tr>
<td>Smaller tags, only a single comparator</td>
<td>Tag is all the bits except block and byte offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Word offset</th>
<th>Byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for tag compare</td>
<td>Selects the set</td>
<td>Selects the word in the block</td>
<td></td>
</tr>
</tbody>
</table>

Word offset

Byte offset
Total Cache Capacity =

Associativity × # of sets × block_size

Bytes = blocks/set × sets × 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 =

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

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

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

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

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

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

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

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

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

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

\[ = \text{address\_size} - (\log_2(SB) - 1) \]
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”)

26
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

- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)
Sources of Cache Misses (3 C’s)

- **Compulsory** (cold start, first reference):
  - 1\textsuperscript{st} access to a block, not a lot you can do about it.
    - If running billions of instructions, compulsory misses are insignificant

- **Capacity**:
  - Cache cannot contain all blocks accessed by the program
    - Misses that would not occur with infinite cache

- **Conflict** (collision):
  - Multiple memory locations mapped to same cache set
    - Misses that would not occur with ideal fully associative cache