CPS104
Computer Organization and Programming
Lecture 14: The Cache

Robert Wagner
Outline of Today’s Lecture

- The Memory Hierarchy
- Direct Mapped Cache
- Two-Way Set Associative Cache
- Fully Associative cache
- Replacement Policies
- Write Strategies
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

- Today’s Topic: Memory System
The Simplest Cache: Direct Mapped Cache

- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index

- Which one should we place in the cache?
- How can we tell which one is in the cache?
Direct Mapped Cache (Cont.)

For a Cache of $2^M$ bytes with block size of $2^L$ bytes

- There are $2^{M-L}$ cache blocks,
- Lowest $L$ bits of the address are Byte Select bits
- Next $(M - L)$ bits are the Cache-Index.
- The last $(32 - M)$ bits are the Tag bits.

<table>
<thead>
<tr>
<th>32-M bits Tag</th>
<th>M-L bits Cache Index</th>
<th>L bits Byte Select</th>
</tr>
</thead>
</table>

Data Address
Example: 1-KB Cache with 32B blocks:

Cache Index = (<Address> Mod (1024))/ 32

Byte Select = <Address> Mod (32) = <Address> & 0x1F

Tag = <Address> / (1024) = <Address> >> 10

<table>
<thead>
<tr>
<th>22 bits Tag</th>
<th>5 bits Cache Index</th>
<th>5 bits Byte Select</th>
</tr>
</thead>
</table>

Address

Valid bit

Cache Tag

22 bits

Direct Mapped Cache Data

32-byte block

<table>
<thead>
<tr>
<th>Byte 31</th>
<th>Byte 30</th>
<th>...</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
</table>

1K = 2^{10} = 1024

2^5 = 32
Example: 1KB Direct Mapped Cache with 32B Blocks

For a 1024 ($2^{10}$) byte cache with 32-byte blocks:

- The uppermost $22 = (32 - 10)$ address bits are the Cache Tag
- The lowest 5 address bits are the Byte Select (Block Size = $2^5$)
- The next 5 address bits (bit5 - bit9) are the Cache Index

![Diagram of cache memory organization](image-url)
Example: 1K Direct Mapped Cache

Cache Tag: 0x0002fe

Cache Index: 0x00

Byte Select: 0x00

Valid Bit: 0

Cache Data:
- Byte 0: 0x00
- Byte 1: 0x00
- Byte 31: 0x00
- Byte 63: 0x00
- Byte 99: 0x00
- Byte 1023: 0x00

Cache Miss
Example: 1K Direct Mapped Cache

Cache Tag

31
0x0002fe

9
0x00

4
0x00

Valid Bit

Cache Tag

1
0x0002fe

1
0x000050

1
0x004440

::

Cache Data

New Block of data

Byte 63  **  Byte 33  Byte 32

Byte 1023  **  Byte 992

Byte Select
Example: 1K Direct Mapped Cache

Cache Tag

<table>
<thead>
<tr>
<th>31</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00050</td>
<td>0x01</td>
<td>0x08</td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Cache Data

<table>
<thead>
<tr>
<th>Byte 31</th>
<th>**</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x002fe</td>
<td>**</td>
<td>0x00050</td>
<td>0x004440</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 63</th>
<th>**</th>
<th>Byte 33</th>
<th>Byte 32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>**</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 1023</th>
<th>**</th>
<th>Byte 992</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>**</td>
<td></td>
</tr>
</tbody>
</table>

Cache Hit

<table>
<thead>
<tr>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
</tr>
</tbody>
</table>

©RW Fall 2000
Example: 1K Direct Mapped Cache

Cache Tag

<table>
<thead>
<tr>
<th>31</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x002450</td>
<td>0x02</td>
<td>0x04</td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x002fe</td>
<td>0x00050</td>
<td>0x004440</td>
</tr>
</tbody>
</table>

Cache Data

<table>
<thead>
<tr>
<th>31</th>
<th>63</th>
<th>33</th>
<th>32</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>**</td>
<td>**</td>
<td>**</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Byte Select

<table>
<thead>
<tr>
<th>1023</th>
<th>992</th>
</tr>
</thead>
</table>

Cache Miss
Example: 1K Direct Mapped Cache

Cache Tag: 0x002450
Cache Index: 0x02
Byte Select: 0x04

Valid Bit:
- 1
- 1
- 1
- :...

Cache Data:
- Byte 31: **
- Byte 63: **
- Byte 1023: **

New Block of data:
- Byte 0: 0
- Byte 1: 0
- Byte 32: 0
- Byte 33: 0
- Byte 992: 0
- Byte 1023: **
Block Size Tradeoff

° In general, larger block sizes take advantage of spatial locality BUT:
  • Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  • If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks

° In general, Average Access Time:
  • Hit Time \times (1 - Miss Rate) + Miss Penalty \times Miss Rate
A N-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operating in parallel

- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result
Advantages of Set associative cache

° Higher Hit rate for the same cache size

° Allows 2 independently placed memory blocks to be in the cache at once (E.G.: Array pieces)

° Fewer Conflict Misses.

° Can have a larger cache but keep the index smaller (So cache index does not overlap virtual page number)
Disadvantage of Set Associative Cache

° N-way Set Associative Cache versus Direct Mapped Cache:
  • N comparators vs. 1
  • Extra MUX delay for the data
  • Data comes AFTER Hit/Miss decision and set selection

° In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  • Possible to assume a hit and continue. Recover later if miss.
And yet Another Extreme Example: Fully Associative cache

- Fully Associative Cache -- push the set associative idea to its limit!
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 32B blocks, we need N 27-bit comparators

- By definition: Conflict Miss = 0 for a fully associative cache
Sources of Cache Misses

° Compulsory (cold start or process migration, first reference): first access to a block
  • “Cold” fact of life: not a whole lot you can do about it

° Conflict (collision):
  • Multiple memory locations mapped to the same cache location
  • Solution 1: increase cache size
  • Solution 2: increase associativity

° Capacity:
  • Cache cannot contain all blocks accessed by the program between accesses to the same block
  • Solution 1: increase cache size
  • Solution 2: change the program

° Invalidation: other process (e.g., I/O) updates memory
## Sources of Cache Misses

<table>
<thead>
<tr>
<th></th>
<th>Direct Mapped</th>
<th>N-way Set Associative</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>Big</td>
<td>Medium</td>
<td>Small</td>
</tr>
<tr>
<td>Compulsory Miss</td>
<td>Same</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>Conflict Miss</td>
<td>High</td>
<td>Medium</td>
<td>Zero</td>
</tr>
<tr>
<td>Capacity Miss</td>
<td>Low(er)</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Invalidation Miss</td>
<td>Same</td>
<td>Same</td>
<td>Same</td>
</tr>
</tbody>
</table>

**Note:**
If you are going to run “billions” of instruction, Compulsory Misses are insignificant.
The Need to Make a Decision!

° Direct Mapped Cache:
  • Each memory location can only map to 1 cache location
  • No need to make any decision :-)  
    - Current item replaced the previous item in that cache location

° N-way Set Associative Cache:
  • Each memory location has a choice of N cache locations

° Fully Associative Cache:
  • Each memory location can be placed in ANY cache location

° Cache miss in an N-way Set Associative or Fully Associative Cache:
  • Bring in new block from memory
  • Throw out a cache block to make room for the new block
  • We need to make a decision on which block to throw out!
Cache Block Replacement Policy Choices

° Random Replacement:
  • Hardware randomly selects a cache item and throws it out

° Least Recently Used:
  • Hardware keeps track of the access history
  • Replace the entry that has not been used for the longest time.
  • For two way set associative cache one needs one bit for LRU replacement.

° Example of a Simple “Pseudo” Least Recently Used Implementation:
  • Assume 64 Fully Associative Entries
  • Hardware replacement pointer points to one cache entry
  • Whenever an access is made to the entry the pointer points to:
    - Move the pointer to the next entry
  • Otherwise: do not move the pointer
  • NOT true LRU -- Why?

<table>
<thead>
<tr>
<th>Entry 0</th>
<th>Entry 1</th>
<th>...</th>
<th>Entry 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement Pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©RW Fall 2000
Cache Write Policy: Write Through versus Write Back

- Cache read is much easier to handle than cache write:
  - Instruction cache is much easier to design than data cache

- Cache write:
  - How do we keep data in the cache and memory consistent?

- Two options (decision time again :-)
  - Write Back: write to cache only. Write the cache block to memory when that cache block is being replaced on a cache miss.
    - Need a “dirty bit” for each cache block
    - Greatly reduces the memory bandwidth requirement
    - Control can be complex
  - Write Through: write to cache and memory at the same time.
    - What!!! How can this be? Isn’t memory too slow for this?
A Write Buffer is needed between the Cache and Memory
- Processor: writes data into the cache and the write buffer
- Memory controller: writes contents of the buffer to memory

Write buffer is just a FIFO:
- Typical number of entries: 4
- Works fine if: Store frequency (w.r.t. time) \(<\) 1 / DRAM write cycle

Memory system designer’s nightmare:
- Store frequency (w.r.t. time) \(>\) 1 / DRAM write cycle
- Write buffer saturation
Write Buffer Saturation

° Store frequency (w.r.t. time) > 1 / DRAM write cycle
  • If this condition exists for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - The CPU Cycle Time will slow to DRAM Write Cycle Time

° Solution for write buffer saturation:
  • Use a write back cache
  • Install a second level (L2) cache:
  • store compression
Write Allocate versus Not Allocate

- Assume: a 16-bit write to memory location 0x0 causes a miss
  - Do we read the block into the cache?
    - Yes: Write Allocate
    - No: Write Not Allocate

### Cache Index

<table>
<thead>
<tr>
<th>Ex: 0x00</th>
<th>Ex: 0x00</th>
</tr>
</thead>
</table>

### Cache Data

<table>
<thead>
<tr>
<th>Byte 31</th>
<th>**</th>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 63</td>
<td>**</td>
<td>Byte 33</td>
<td>Byte 32</td>
</tr>
<tr>
<td>Byte 1023</td>
<td>**</td>
<td>Byte 992</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dirty-bit</th>
<th>Valid Bit</th>
<th>Cache Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x00</td>
</tr>
</tbody>
</table>

| Byte 63 | ** | Byte 33| Byte 32|
| Byte 1023 | ** | Byte 992|

©RW Fall 2000
Four Questions for Memory Hierarchy Designers

° Q1: Where can a block be placed in the upper level? (Block placement)

° Q2: How is a block found if it is in the upper level? (Block identification)

° Q3: Which block should be replaced on a miss? (Block replacement)

° Q4: What happens on a write? (Write strategy)
Summary:

° The Principle of Locality:
  • Program accesses a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space

° Three Major Categories of Cache Misses:
  • Compulsory Misses: sad fact of life. Example: cold start misses.
  • Conflict Misses: increase cache size and/or associativity.
    Nightmare Scenario: ping pong effect!
  • Capacity Misses: increase cache size

° Write Policy:
  • Write Through: need a write buffer. Nightmare: WB saturation
  • Write Back: control can be complex