Cache Memory

Computer Science 104

Administrivia

- Midterm II Monday (covers up through pipelining)
- Homework #5

Reading
- Chapter 6.1-6.4
Outline of Today’s Lecture

° Review
° The Memory Hierarchy
° Direct-mapped Cache
° Two-Way Set Associative Cache
° Fully Associative cache
° Replacement Policies
° Write Strategies

Issues for Memory Systems

° Capacity/Size
° Cost
  • What technology is cheap?
° Performance
  • What technology is fast?
° Ease of Use
  • How much do programmers have to worry about it?
**Cache**

- What is a cache?
- What is the motivation for a cache?
- Why do caches work?
- How do caches work?

---

**The Motivation for Caches**

- **Motivation:**
  - Large memories (DRAM) are *slow*
  - Small memories (SRAM) are *fast*
- Make the *average access time* small by:
  - Servicing most accesses from a small, fast memory.
- Reduce the *bandwidth* required of the large memory
Levels of the Memory Hierarchy

<table>
<thead>
<tr>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100s Bytes</td>
<td>~1 ns</td>
</tr>
<tr>
<td>Cache</td>
<td>K Bytes</td>
<td>1-100 ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>M Bytes</td>
<td>100ns-1us</td>
</tr>
<tr>
<td>Disk</td>
<td>G Bytes</td>
<td>ms ~10^{-4}</td>
</tr>
<tr>
<td>Tape</td>
<td>infinite</td>
<td>sec/min</td>
</tr>
</tbody>
</table>

The Principle of Locality:

- Program access a relatively small portion of the address space at any instant of time.
- Example: 90% of time in 10% of the code

Two Different Types of Locality:

- **Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
- **Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.
Memory Hierarchy: Principles of Operation

- At any given time, data is copied between only 2 adjacent levels:
  - Upper Level (Cache): the one closer to the processor
    - Smaller, faster, and uses more expensive technology
  - Lower Level (Memory): the one further away from the processor
    - Bigger, slower, and uses less expensive technology

- Block:
  - The minimum unit of information that can either be present or not present in the two level hierarchy

Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss

- Miss: data needs to be retrieved from a lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty = Time to replace a block in the upper level + Time to deliver the block the processor

- Hit Time << Miss Penalty
Direct Mapped Cache

- Direct Mapped cache: array of fixed size frames.
- Each frame holds consecutive bytes of main memory data (block).
- The Tag Array holds the Block Memory Address.
- A valid bit associated with each cache block tells if the data is valid.

- Cache Index: The location of a block (and its tag) in the cache.
- Block Offset: The byte location in the cache block.

Cache-Index = (Address Mod (Cache_Size))/ Block_Size
Block-Offset = Address Mod (Block_Size)
Tag = Address / (Cache_Size)

The Simplest Cache: Direct Mapped Cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

4 Byte Direct Mapped Cache with 1-byte blocks

- 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 Block-Offset 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</th>
<th>Tag</th>
<th>M-L bits</th>
<th>Cache Index</th>
<th>L bits block offset</th>
</tr>
</thead>
</table>

Data Address

Example: 1-KB Cache with 32B blocks:

**Cache Index** = \(<\text{Address}> \mod (1024)) / 32

**Block-Offset** = \(<\text{Address}> \mod (32)

**Tag** = \(<\text{Address}> / (1024)

<table>
<thead>
<tr>
<th>22 bits</th>
<th>Tag</th>
<th>5 bits</th>
<th>Cache Index</th>
<th>5 bits block offset</th>
</tr>
</thead>
</table>

Address

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Cache Tag</th>
<th>22 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Direct Mapped Cache Data

<table>
<thead>
<tr>
<th>Byte 31</th>
<th>Byte 30</th>
<th>. . . .</th>
<th>Byte 4</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-byte block

32 cache blocks

$1K = 2^{10} = 1024$

$2^5 = 32$
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**

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Example: 0x50</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit  | Cache Tag  | 0x50        |             |             |
|           |            |             |             |             |

Store as part of the cache “state”

<table>
<thead>
<tr>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 31</td>
</tr>
<tr>
<td>Byte 63</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

|          |          |              |              |
| Byte 1023| ** Byte 992 |              |              |

Example: 1K Direct Mapped Cache

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>9 Cache Index</th>
<th>4 Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002fe</td>
<td>0x00</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Valid Bit  | Cache Tag  | 0xxxxxxx | 0x000050 | 0x004440 |
|           |            |          |          |          |

Cache Miss
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>0x0002fe</td>
<td>0x00</td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit  

Cache Tag

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002fe</td>
<td>0x000050</td>
<td>0x004440</td>
<td>:</td>
</tr>
</tbody>
</table>

Cache Data

New Block of data

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

Byte 1023 | ** | Byte 992 |

Valid Bit  

Cache Tag

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000050</td>
<td>0x000050</td>
<td>0x000050</td>
<td>:</td>
</tr>
</tbody>
</table>

Cache Data

Cache Hit  

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>0x000050</td>
<td>0x01</td>
<td>0x08</td>
<td></td>
</tr>
</tbody>
</table>

Valid Bit  

Cache Tag

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0002fe</td>
<td>0x000050</td>
<td>0x004440</td>
<td>:</td>
</tr>
</tbody>
</table>

Cache Data

Cache Hit  

Byte Select
Example: 1K Direct Mapped Cache

Cache Tag

0x002450

0x02

0x04

Valid Bit

Cache Tag

0x0002fe

0x000050

0x004440

Cache Data

Byte 0

0x0002fe

Byte 63

**

Byte 31

**

Byte 1

0x000050

Byte 32

1

Byte 33

2

Byte 992

31

Byte 1023

**

Cache Miss

New Block of data
**Block Size Tradeoff**

- In general, larger block size 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 x (1 - Miss Rate) + Miss Penalty x 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.
- Fewer **Conflict Misses**.
- Can can have a larger cache but keep the index smaller.

**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

```
<table>
<thead>
<tr>
<th>Cache Tag (27 bits long)</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex: 0x01</td>
<td></td>
</tr>
</tbody>
</table>
```

![Cache Diagram]

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 access by the program
  - Solution: increase cache size

- 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 replaces the previous item in that cache location

° N-way Set Associative Cache:
  • For each memory location have a choice of N cache locations

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

° Cache miss in a 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

° Random Replacement:
  • Hardware randomly selects a cache item and throw 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 (Not Most Recently Used):
  • 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

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 reduce 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?
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

° Caches provide cost effective memory system
° Work by exploiting locality (temporal & spatial)
° Associativity, Blocksize, Capacity (ABCs of caches)
° Know how a cache works
  • Break address into tag,index, block offset
° Know how to draw a block diagram of a cache

Next Time
° Cache Performance and Programming