Cache Organization

Joel Emer  
Computer Science and Artificial Intelligence Laboratory  
M.I.T.

http://www.csg.csail.mit.edu/6.823
**CPU-Memory Bottleneck**

Performance of high-speed computers is usually limited by memory *bandwidth* & *latency*

- **Latency** (time for a single access)
  Memory access time $>>$ Processor cycle time

- **Bandwidth** (number of accesses per unit time)
  if fraction $m$ of instructions access memory,
  $\Rightarrow 1+m$ memory references / instruction
  $\Rightarrow$ CPI = 1 requires $1+m$ memory refs / cycle
Memory Technology

• Early machines used a variety of memory technologies
  – Manchester Mark I used CRT Memory Storage
  – EDVAC used a mercury delay line

• Core memory was first large scale reliable main memory
  – invented by Forrester in late 40s at MIT for Whirlwind project
  – Bits stored as magnetization polarity on small ferrite cores threaded onto 2 dimensional grid of wires

• First commercial DRAM was Intel 1103
  – 1Kbit of storage on single chip
  – charge on a capacitor used to hold value

• Semiconductor memory quickly replaced core in 1970s
  – Intel formed to exploit market for semiconductor memory

• Flash memory
  – Slower, but denser than DRAM. Also non-volatile, but with wearout issues

• Phase change memory (PCM) a possibility for the future
  – Slightly slower, but much denser than DRAM and non-volatile
• Bits stored in 2-dimensional arrays on chip
• Modern chips have around 8 logical banks on each chip
  – each logical bank physically implemented as many smaller arrays
DRAM timing

DRAM Spec:
CL, tRCD, tRP, tRAS, e.g., 9-9-9-24
Processor-DRAM Gap (latency)

Four-issue 2GHz superscalar accessing 100ns DRAM could execute 800 instructions during time for one memory access!
HBM DRAM

AMD Diagram
Little’s Law

Throughput \( (T) = \frac{\text{Number in Flight (N)}}{\text{Latency (L)}} \)

Example:

--- Assume infinite bandwidth memory
--- 100 cycles / memory reference
--- 1 + 0.2 memory references / instruction

\[ \Rightarrow \text{Table size} = 1.2 \times 100 = 120 \text{ entries} \]

120 independent memory operations in flight!
Basic Static RAM Cell

6-Transistor SRAM Cell

- Write:
  1. Drive bit lines (bit=1, bitbar=0)
  2. Select row
- Read:
  1. Precharge bit and bitbar to Vdd
  2. Select row
  3. Cell pulls one line low
  4. Column sense amp detects difference between bit & bitbar

Often replaced with pullup to save area
Multilevel Memory

Strategy: Reduce average latency using small, fast memories called caches.

Caches are a mechanism to reduce memory latency based on the empirical observation that the patterns of memory references made by a processor are often highly predictable:

```
<table>
<thead>
<tr>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>96</td>
</tr>
<tr>
<td>loop: ADD r2, r1, r1</td>
</tr>
<tr>
<td>SUBI r3, r3, #1</td>
</tr>
<tr>
<td>BNEZ r3, loop</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>108</td>
</tr>
<tr>
<td>112</td>
</tr>
</tbody>
</table>
```

L03-10
Typical Memory Reference Patterns

- Instruction fetches
- Stack accesses
- Data accesses

Address

Time

n loop iterations

subroutine call

argument access

subroutine return

vector access

scalar accesses

Sanchez & Emer

February 10, 2016

http://www.csg.csail.mit.edu/6.823
Common Predictable Patterns

Two predictable properties of memory references:

- **Temporal Locality**: If a location is referenced it is likely to be referenced again in the near future.

- **Spatial Locality**: If a location is referenced it is likely that locations near it will be referenced in the near future.
Memory Hierarchy

- **size:** Register << SRAM << DRAM  
- **latency:** Register << SRAM << DRAM  
- **bandwidth:** on-chip >> off-chip

On a data access:

- **hit** (data ∈ fast memory) ⇒ low latency access
- **miss** (data ∉ fast memory) ⇒ long latency access \((DRAM)\)
Management of Memory Hierarchy

• Small/fast storage, e.g., registers
  - Address usually specified in instruction
  - Generally implemented directly as a register file
    • but hardware might do things behind software’s back, e.g., stack management, register renaming

• Large/slower storage, e.g., memory
  - Address usually computed from values in register
  - Generally implemented as a cache hierarchy
    • hardware decides what is kept in fast memory
    • but software may provide “hints”, e.g., don’t cache or prefetch
Inside a Cache

How many bits are needed in tag?

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Data Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6848</td>
<td></td>
<td></td>
</tr>
<tr>
<td>416</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enough to uniquely identify block
Cache Algorithm (Read)

Look at Processor Address, search cache tags to find match. Then either

- Found in cache a.k.a. HIT
  - Return copy of data from cache

- Not in cache a.k.a. MISS
  - Read block of data from Main Memory
    - Wait ...
  - Return data to processor and update cache

Q: Which line do we replace?
Direct-Mapped Cache

What is a bad reference pattern?

Strided at size of cache
Direct Map Address Selection

higher-order vs. lower-order address bits

Why might this be undesirable?

Spatially local blocks conflict
Hashed Address Selection

What are the tradeoffs of hashing?

**Good:** Regular strides don’t conflict
**Bad:** Hash adds latency
Tag is larger
2-Way Set-Associative Cache

![Diagram of a 2-Way Set-Associative Cache]

- **Tag**: Represents the unique identifier of a block within a set.
- **Index**: Determines the set in which the block resides.
- **Block Offset**: Indicates the position of the data within the block.

The diagram shows how tags and data blocks are matched and how a hit or miss is determined based on the comparison of tags and the selection of the appropriate data block.
Placement Policy

Memory

Block Number

Set Number

Cache

Direct Mapped

(2-way) Set Associative

Fully Associative

block 12 can be placed only into block 4

(12 mod 8)

(12 mod 4)

(12 mod 4)
Fully Associative Cache
Improving Cache Performance

Average memory access time =
    Hit time + Miss rate x Miss penalty

To improve performance:
• reduce the hit time
• reduce the miss rate (e.g., larger, better policy)
• reduce the miss penalty (e.g., L2 cache)

What is the simplest design strategy?

Biggest cache that doesn’t increase hit time past 1-2 cycles
(approx 8-32KB in modern technology)
[design issues more complex with out-of-order superscalar processors]
Causes for Cache Misses

• **Compulsory:**
  first-reference to a block *a.k.a.* cold start misses
  - misses that would occur even with infinite cache

• **Capacity:**
  cache is too small to hold all data the program needs
  - misses that would occur even under perfect placement & replacement policy

• **Conflict:**
  misses from collisions due to block-placement strategy
  - misses that would not occur with full associativity
### Effect of Cache Parameters on Performance

<table>
<thead>
<tr>
<th></th>
<th>Larger capacity cache</th>
<th>Higher associativity cache</th>
<th>Larger block size cache *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory misses</td>
<td>=</td>
<td>=</td>
<td>↓</td>
</tr>
<tr>
<td>Capacity misses</td>
<td>↓</td>
<td>=</td>
<td>↓</td>
</tr>
<tr>
<td>Conflict misses</td>
<td>↓</td>
<td>↓</td>
<td>?</td>
</tr>
<tr>
<td>Hit latency</td>
<td>↑</td>
<td>↑</td>
<td>=</td>
</tr>
<tr>
<td>Miss latency</td>
<td>=</td>
<td>=</td>
<td>↑↓</td>
</tr>
</tbody>
</table>

* Assume substantial spatial locality
Block-level Optimizations

• Tags are too large, i.e., too much overhead
  – Simple solution: Larger blocks, but miss penalty could be large.

• Sub-block placement (aka sector cache)
  – A valid bit added to units smaller than the full block, called sub-blocks
  – Only read a sub-block on a miss
  – *If a tag matches, is the word in the cache?*

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>204</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Replacement Policy

Which block from a set should be evicted?

- Random

- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way

- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches

- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block or blocks

- One-bit LRU
  - Each way represented by a bit. Set on use, replace first unused.
Re-reference interval prediction

References:

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>F</td>
<td>E</td>
<td>F</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>F</td>
</tr>
</tbody>
</table>

**Tag**

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>A</td>
<td>F</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RRI**

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Best candidate?**

- A

**RRIP** bit LRU

- 2 3 0 1
Multiple replacement policies

Use the best replacement policy for a program

How do we decide which policy to use?

- Policy A
- Policy B

Sets

Cache

Miss

0: Policy A Missed
1: Policy B Missed

Counter

+1

-1

0 > 0

0: Policy A
1: Policy B

February 10, 2016

http://www.csg.csail.mit.edu/6.823

Sanchez & Emer
Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
  - Each level can have longer latency than level above
  - So, increase sizes of cache at each level

Metrics:

Local miss rate = misses in cache/ accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions
Inclusion Policy

• **Inclusive multilevel cache:**
  - Inner cache holds copies of data in outer cache
  - External access need only check outer cache
  - Most common case

• **Exclusive multilevel caches:**
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

Why choose one type or the other?
Victim Caches (HP 7200)

Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines:
- First look up in direct mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim ->?

Fast hit time of direct mapped but with reduced conflict misses
Way Predicting Caches
(MIPS R10000 L2 cache)

- Use processor address to index into way prediction table
- Look in predicted way at given index, then:
  - **HIT**: Return copy of data from cache
  - **MISS**: Look in other way
    - **MISS**: Read block of data from next level of cache
    - **SLOW HIT**: Return copy of data from cache (change entry in prediction table)
Way Predicting Instruction Cache (Alpha 21264-like)
Typical memory hierarchies

(a) Memory hierarchy for server

- CPU
  - Registers
  - Level 1 Cache: Reference
    - Size: 1000 bytes
    - Speed: 300 ps

- L1 Cache
  - Size: 64 KB
  - Speed: 1 ns

- L2 Cache
  - Size: 256 KB
  - Speed: 3–10 ns

- L3 Cache
  - Size: 2–4 MB
  - Speed: 10–20 ns

- Memory
  - Size: 4–16 GB
  - Speed: 50–100 ns

- Disk Storage
  - Size: 4–16 TB
  - Speed: 5–10 ms

(b) Memory hierarchy for a personal mobile device

- CPU
  - Registers
  - Level 1 Cache: Reference
    - Size: 500 bytes
    - Speed: 500 ps

- L1 Cache
  - Size: 64 KB
  - Speed: 2 ns

- L2 Cache
  - Size: 256 KB
  - Speed: 10–20 ns

- Memory
  - Size: 256–512 MB
  - Speed: 50–100 ns

- Storage
  - Size: 4–8 GB
  - Speed: 25–50 us
Thank you!

Next lecture – virtual memory