The Memory Hierarchy

Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab
M.I.T.
# Memory Technologies

Different technologies have vastly different tradeoffs.

Size is a **fundamental limit**, even setting cost aside:
- Small + low latency or
- Large + high-latency

Can we get best of both worlds? (large, fast, cheap)

<table>
<thead>
<tr>
<th>Memory Technology</th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>100s of bits</td>
<td>20 ps</td>
<td>$$$$$</td>
</tr>
<tr>
<td>SRAM</td>
<td>~10 KB-10 MB</td>
<td>1-10 ns</td>
<td>~$1000</td>
</tr>
<tr>
<td>DRAM</td>
<td>~10 GB</td>
<td>80 ns</td>
<td>~$10</td>
</tr>
<tr>
<td>Flash*</td>
<td>~100 GB</td>
<td>100 us</td>
<td>~$1</td>
</tr>
<tr>
<td>Hard disk*</td>
<td>~1 TB</td>
<td>10 ms</td>
<td>~$0.1</td>
</tr>
</tbody>
</table>

* non-volatile (retains contents when powered off)

---

April 4, 2019
Exposed Memory Hierarchy

- Attach a variety of storage alternatives (registers, SRAM, and DRAM) of varying sizes to the CPU
- Tell programmers: “Use them cleverly”
- Implies that you either:
  1. Modify ISA to provide different instructions for accessing the different memory elements.
  2. Allocate specific regions of address space for each type of memory.
- What happens if you want to increase the size of your SRAM? Not really useful in practice
Implicit Memory Hierarchy

- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns
- CPU effectively sees large, fast memory if values are found in cache most of the time.
Why Caches Work

- Two predictable properties of memory accesses:
  - **Temporal locality**: If a location has been accessed recently, it is likely to be accessed (reused) in the near future.
  - **Spatial locality**: If a location has been accessed recently, it is likely that nearby locations will be accessed in the near future.
Typical Memory Access Patterns

- Array accesses
- Local variable accesses
- Procedure calls
- Loop

Address vs. Time

Data vs. Stack

Code vs. Time
Caches

- Cache: A small, interim storage component that transparently retains (caches) data from recently accessed locations

- Processor sends accesses to cache. Two options:
  - **Cache hit**: Data for this address in cache, returned quickly
  - **Cache miss**: Data not in cache
    - Fetch data from memory, send it back to processor
    - Retain this data in the cache (replacing some other data)

  Processor must deal with variable access-time of memory
Cache Metrics

- Hit Ratio: \[ HR = \frac{\text{hits}}{\text{hits} + \text{misses}} = 1 - MR \]

- Miss Ratio: \[ MR = \frac{\text{misses}}{\text{hits} + \text{misses}} = 1 - HR \]

- Average Memory Access Time (AMAT):
  \[ AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty} \]

Cache design is all about reducing AMAT
Example: How High of a Hit Ratio?

AMAT without a cache = 100 cycles
Latency with cache: Hit = 4 cycles; Miss = 104 cycles
What hit ratio do we need to break even?

\[ 100 = 4 + (1 - HR) \times 100 \Rightarrow HR = 4\% \]

AMAT for different hit ratios:
- HR=50\% \Rightarrow AMAT = 4 + (1 - 0.50) \times 100 = 54
- HR=90\% \Rightarrow AMAT = 4 + (1 - 0.90) \times 100 = 14
- HR=99\% \Rightarrow AMAT = 4 + (1 - 0.99) \times 100 = 5

With high HR caches can dramatically improve AMAT

should be easy to achieve

Can we achieve such high HR?
Basic Cache Algorithm (Reads)

Q: How do we “search” the cache?

On reference to Mem[X], look for X among cache tags

HIT: \( X = \text{Tag}(i) \) for some cache line \( i \)

Return Data(i)

MISS: X not found in Tag of any cache line

1. Read Mem[X]
2. Return Mem[X]
3. Select a line \( k \) to hold Mem[X]
4. Write Tag(k) = X, Data(k) = Mem[X]
Direct-Mapped Caches

- Each word in memory maps into a single cache line
- Access (for cache with $2^w$ lines):
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, HIT
- Example 8-line direct-mapped cache:
Example: Direct-Mapped Caches

64-line direct-mapped cache \(\rightarrow\) 64 indexes \(\rightarrow\) 6 index bits

**Read Mem[0x400C]**

```
0100 0000 0000 1100
```

- **TAG:** 0x40
- **INDEX:** 0x3
- **OFFSET:** 0x0

**HIT, DATA 0x42424242**

Would 0x4008 hit?

**INDEX:** 0x2 \(\rightarrow\) tag mismatch
\(\rightarrow\) **MISS**

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data’s address
Further exploiting spatial locality

- Store multiple words per data line
  - Main advantage: Exploit spatial locality
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer lines in the cache (more conflicts)

- Example: 4-word line, 16-word direct-mapped cache

---

32-bit BYTE address

- Tag bits: 26 (=32-4-2)
- Index bits: 2 (4 indexes)
- Byte offset bits: 2 (ensure word alignment)

Line offset bits: 2 (line size = 4)
Line Size Tradeoffs

- Larger line sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the line from memory
  - Can increase the average hit time and miss ratio
- \( \text{AMAT} = \text{HitTime} + \text{MissPenalty} \times \text{MissRatio} \)
Write Policy

1. Write-through: CPU writes are cached, but also written to main memory immediately; Memory always holds current contents

2. Write-back: CPU writes are cached, but not written to main memory until we replace the line. Memory contents can be “stale”
   - Upon replacement, a modified cache line must first be written back to memory before loading the new cache line
   - To avoid unnecessary writebacks, a Dirty bit is added to each cache line to indicate if the value has been modified since it was loaded from memory

3. No cache write on a Write-miss: On a cache miss, write is sent directly to the memory without a cache write

Write-back is the most commonly used policy, because it saves cache-memory bandwidth
Example: Cache Write-Hit

16-line direct-mapped cache → 4 index bits
line size = 4 → 2 line offset bits
Write Policy = Write Back
Write: 0x09 to 0x4818

Tag: 0x48
Index: 0x1
line Off: 2
Byte Off: 0x0

D=1: cache contents no longer match main memory so write back line to memory upon replacement
Example: Cache Write-Miss

1. Tags don’t match -> Miss
   - **D=1**: Write cache line 1 (tag = 0x280: addresses 0x28010-0x2801C) back to memory
   - If **D=0**: Don’t need to write line back to memory.

2. Load line (tag = 0x48: addresses 0x4810-0x481C) from memory

3. Write 0x09 to 0x4818 (line offset 2), set D=1.
## Direct-Mapped Cache Problem: Conflict Misses

### Loop A:
- **Code at 1024**, data at **37**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/ Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>HIT</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>HIT</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>HIT</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>HIT</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume:
- 1024-line DM cache
- line size = 1 word
- Consider looping code, in steady state
- Assume WORD, not BYTE, addressing

Inflexible mapping (each address can only be in one cache location) \(\rightarrow\) Conflict misses!

### Loop B:
- **Code at 1024**, data at **2048**

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Cache Line index</th>
<th>Hit/ Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>2049</td>
<td>1</td>
<td>MISS</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>2050</td>
<td>2</td>
<td>MISS</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
N-way Set-Associative Cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- **Nomenclature:**
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = # ways = “set associativity” (e.g. 4-way → 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel

- **Fully-associative cache:** Number of ways = Number of lines
  - Any address can be in any line → No conflict misses, but expensive
**Associativity Implies Choices**

**Issue: Replacement Policy**

- **Direct-mapped**
  - Compare addr with only one tag
  - Location A can be stored in exactly one cache line

- **N-way set-associative**
  - Compare addr with N tags simultaneously
  - Location A can be stored in exactly one set, but in any of the N cache lines belonging to that set

- **Fully associative**
  - Compare addr with each tag simultaneously
  - Location A can be stored in any cache line
Replacement Policies

- **Least Recently Used (LRU):** Replace the line that was accessed furthest in the past
  - Works well in practice
  - Need to keep ordered list of $N$ items $\rightarrow N!$ orderings
    $\rightarrow O(\log_2 N!) = O(N \log_2 N)$ “LRU bits” + complex logic
  - Caches often implement cheaper approximations of LRU

- Other policies:
  - First-In, First-Out (least recently replaced)
  - Random: Choose a candidate at random
    - Not very good, but does not have adversarial access patterns
Summary: Cache Tradeoffs

Caches allow memory to appear like a large, fast, and cheap memory.

\[
AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty}
\]

Design tradeoffs can be made in cache size, line size, associativity, replacement policy, write policy.

Thank you!
This material will be included in Quiz 2

Next lecture: Implementing Caches
Take home

- Compare the hit rates for the following 3 caches: Direct Mapped, 2-Way Set Associative, and Fully Associative. Assume each has 8 (4 byte) words. Assume that the access pattern is repeatedly: 0, 16, 4, 36.