The Memory Hierarchy

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

- Technologies have vastly different tradeoffs between capacity, latency, bandwidth, energy, and cost.
  - ... and thus different applications

<table>
<thead>
<tr>
<th></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)
## Summary: Memory Technologies

- Different technologies have vastly different tradeoffs
- Size is a **fundamental limit**, even setting cost aside:
  - Small + low latency, high bandwidth, low energy, or
  - Large + high-latency, low bandwidth, high energy
- Can we get best of both worlds? (large, fast, cheap)

<table>
<thead>
<tr>
<th></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>
The Memory Hierarchy

Want large, fast, and cheap memory, but...
Large memories are slow (even if built with fast components)
Fast memories are expensive

Solution: Use a hierarchy of memories with different tradeoffs to fake a large, fast, cheap memory

<table>
<thead>
<tr>
<th></th>
<th>Fastest</th>
<th>Slowest</th>
<th>Fast</th>
<th>Large</th>
<th>Cheap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Capacity</td>
<td>Cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
</tr>
</tbody>
</table>

≈
Memory Hierarchy Interface

Approach 1: Expose Hierarchy
- Registers, SRAM, DRAM, Flash, Hard Disk each available as storage alternatives
- Tell programmers: "Use them cleverly"

Approach 2: Hide Hierarchy
- Programming model: Single memory, single address space
- Machine transparently stores data in fast or slow memory, depending on usage patterns
Typical Memory Access Patterns

- **address**
  - array accesses
  - local variable accesses
  - procedure calls

- **data**
  - loop

- **stack**

- **code**

**time**

October 23, 2018
Common Predictable Patterns

- Two predictable properties of memory accesses:
  - **Temporal locality**: If a location has been accessed recently, it is likely to be accessed (reused) soon.
  - **Spatial locality**: If a location has been accessed recently, it is likely that nearby locations will be accessed soon.
Caches

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

  ![Diagram showing CPU, Cache, Main Memory, Address, and Data connections.]

- 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 memory access time.
A Typical Memory Hierarchy

Computers use many levels of caches:

<table>
<thead>
<tr>
<th>On the CPU</th>
<th>Access time</th>
<th>Capacity</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1 cycle</td>
<td>1 KB</td>
<td>Software/Compiler</td>
</tr>
<tr>
<td>Level 1 Cache</td>
<td>2-4 cycles</td>
<td>32 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>Level 2 Cache</td>
<td>10 cycles</td>
<td>256 KB</td>
<td>Hardware</td>
</tr>
<tr>
<td>Level 3 Cache</td>
<td>40 cycles</td>
<td>10 MB</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other chips and devices</th>
<th>Access time</th>
<th>Capacity</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory</td>
<td>200 cycles</td>
<td>10 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>Flash Drive</td>
<td>10-100us</td>
<td>100 GB</td>
<td>Software/OS</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>10ms</td>
<td>1 TB</td>
<td>Software/OS</td>
</tr>
</tbody>
</table>

October 23, 2018
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):** 
  \[ AMAT = \text{HitTime} + \text{MissRatio} \times \text{MissPenalty} \]

- Goal of caching is to improve AMAT
- Formula can be applied recursively in multi-level hierarchies:

  \[ AMAT = \text{HitTime}_{L_1} + \text{MissRatio}_{L_1} \times AMAT_{L_2} = \]

  \[ AMAT = \text{HitTime}_{L_1} + \text{MissRatio}_{L_1} \times (\text{HitTime}_{L_2} + \text{MissRatio}_{L_2} \times AMAT_{L_3}) = \ldots \]
Example: How High of a Hit Ratio?

What hit ratio do we need to break even? (Main memory only: AMAT = 100)

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

What hit ratio do we need to achieve AMAT = 5 cycles?

\[ 5 = 4 + (1 - HR) \times 100 \Rightarrow HR = 99\% \]
Basic Cache Algorithm (Reads)

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

**HIT:** X = Tag(i) for some cache line i
Return Data(i)

**MISS:** X not found in Tag of any cache line
Read Mem[X] Return Mem[X]
Select a line k to hold Mem[X]
Write Tag(k) = X, Data(k) = Mem[X]

Q: How do we “search” the cache?
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 → 64 indexes → 6 index bits

Read Mem[0x400C]

0100 0000 0000 1100

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

HIT, DATA 0x42424242

Would 0x4008 hit?

INDEX: 0x2 → tag mismatch → MISS

Part of the address (index bits) is encoded in the location
Tag + Index bits unambiguously identify the data’s address
Block Size

- Take advantage of spatial locality: Store multiple words per data block
  - Another advantage: Reduces size of tag memory!
  - Potential disadvantage: Fewer blocks in the cache
- Example: 4-block, 16-word direct-mapped cache

32-bit BYTE address

Tag bits: 26 (=32-4-2)

Index bits: 2 (4 indexes)

Block offset bits: 4 (16 bytes/block)
Block Size Tradeoffs

- Larger block sizes...
  - Take advantage of spatial locality
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio
- \[ \text{AMAT} = \text{HitTime} + \text{MissPenalty} \times \text{MissRatio} \]
Direct-Mapped Cache Problem: Conflict Misses

Assume:
- 1024-line DM cache
- Block 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) → Conflict misses!

<table>
<thead>
<tr>
<th>Loop A: Code at 1024, data at 37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Address</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>1025</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop B: Code at 1024, data at 2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Address</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
</tr>
<tr>
<td>1025</td>
</tr>
<tr>
<td>2049</td>
</tr>
<tr>
<td>1026</td>
</tr>
<tr>
<td>2050</td>
</tr>
<tr>
<td>1024</td>
</tr>
<tr>
<td>2048</td>
</tr>
<tr>
<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: Extreme case with a single set and as many ways as lines
  - Any address can be in any line → No conflict misses, but expensive
N-way Set-Associative Cache

Example: 3-way 8-set cache
Memory Technologies: SRAM and DRAM

NOTE: Demystification, will not be on the quiz
Static RAM (SRAM)

8x6 SRAM array

Address decoder

Drivers

SRAM cell

Wordlines (horizontal)

Bitlines (vertical)

Sense amplifiers

Data in

Address

3

Data out

6

October 23, 2018
SRAM Cell

- A bistable element consisting of two inverters
- Two access transistors

Bistable element (two stable states) stores a single bit

Vdd
GND “1”

GND
Vdd “0”

Transistor (FET) is a voltage-controlled switch

OFF
GND
ON
Vdd
SRAM Read

1. Drivers precharge all bitlines to Vdd (1), and leave them floating

2. Address decoder activates one wordline

3. Each cell in the activated word slowly pulls down one of the bitlines to GND (0)

4. Sense amplifiers sense change in bitline voltages, produce output data
1. Drivers hold bitlines to desired values (Vdd and GND for 1, GND and Vdd for 0)
2. Address decoder activates one wordline
3. Each cell in word is overpowered by the drivers, stores value

Cell transistors are carefully sized so that bitline GND overpowers cell Vdd, but bitline Vdd does not overpower cell GND
Multiported SRAMs

- SRAM so far can do either one read or one write/cycle
- We can do multiple reads and writes with multiple ports by adding one set of wordlines and bitlines per port

- \textit{Cost/bit for N ports?}
  - \textit{Wordlines?} \(N\)
  - \textit{Bitlines?} \(2^N\)
  - \textit{Access transistors?} \(2^N\)

- Wires dominate area \(\rightarrow O(N^2)\) area!
**1T Dynamic RAM (DRAM) Cell**

- Storage capacitor
- 1T DRAM Cell
- Wordline
- Access transistor
- Bitline

✓ ~20x smaller area than SRAM cell → Denser and cheaper!

✗ Problem: Capacitor leaks charge, must be refreshed periodically (~milliseconds)
**DRAM Writes and Reads**

- **Writes**: Drive bitline to Vdd or GND, activate wordline, charge or discharge capacitor.

- **Reads**:  
  1. Precharge bitline to Vdd/2  
  2. Activate wordline  
  3. Capacitor and bitline share charge  
     - If capacitor was discharged, bitline voltage decreases slightly  
     - If capacitor was charged, bitline voltage increases slightly  
  4. Sense bitline to determine if 0 or 1

- **Issue**: **Reads are destructive**! (charge is gone!)  
  - Data must be rewritten to cells at end of read
Thank you!

Next lecture: Implementing Caches