Memory Hierarchy - principles + cache

Prof. Yitzhak Birk
Electrical Engineering Department, Technion
Memory Hierarchy— Motivation, Definitions, Four Questions about Memory Hierarchy
Motivation: the CPU - DRAM Gap

• 1980: no cache in microprocessor;

• 1995 2-level cache, 60% of the transistors on the Alpha 21164 μprocessor
Memory Hierarchy

- The number of CPU cycles to reach memory domain

\[ C = \text{CPU cycles} \]
General Principles

• **Locality** – Memory accesses made by the program are NOT uniformly distributed.
  – *Temporal Locality*: same address referenced again soon
  – *Spatial Locality*: nearby addresses referenced soon

• Locality +
  + smaller HW is faster
  + faster HW is more expensive
  + Amdahl’s law

=> memory hierarchy
CPU registers are the highest level of memory, but are managed by the compiler and/or CPU differently than the rest of the hierarchy.
Terminology

- **Block** – quantum of data within memory hierarchy.
- **Hit** – A block is present at the searched level.
- **Miss** - A block is NOT present at the searched level.
- **Hit time** – the search time of a block (includes block access in case of hit).
- **Miss penalty** - time to replace a block from lower level, including time to provide to CPU.
- **Miss time** = hit time + miss penalty.
- **Hit rate** - fraction of accesses found in that level.
- **Miss rate** = 1 - hit rate.
- **Inclusion** – every block in level j, also resides in levels j+1, j+2, ...
- **Exclusion** – every block in level j, does NOT reside in levels j+1, j+2, ...

**Remarks:**
- Definitions are “floating”, i.e., refer to the current level and the one below it.
- Block sizes in lower (slower) levels are typically power-of-two multiples of those at higher levels.
Graph of Events

Event 1
- $P_{hit\text{-}L1}$

Event 2
- $P_{hit\text{-}L2}$

Event 3
- $P_{hit\text{-}mem}$

Event 4
- $P_{miss\text{-}mem} = \text{page fault}$

$L1$ cache
- $P_{hit\text{-}L1}$
- $P_{miss\text{-}L1}$

$L2$ cache
- $P_{hit\text{-}L2}$
- $P_{miss\text{-}L2}$

Memory
- $P_{hit\text{-}mem}$

Disk
- $P_{miss\text{-}mem}$

$P_{Event1} = P_{hit\text{-}L1}$

$P_{Event2} = P_{miss\text{-}L1} \times P_{hit\text{-}L2}$

$P_{Event3} = P_{miss\text{-}L1} \times P_{miss\text{-}L2} \times P_{hit\text{-}mem}$

$P_{Event4} = P_{miss\text{-}L1} \times P_{miss\text{-}L2} \times P_{miss\text{-}mem}$
Memory System Measures

- **Average memory-access time (AMAT)** = Hit time + Miss rate x Miss penalty (ns or CPU clk cycles)

- **Miss penalty**: time to replace a block from lower level(s), including time to replace in CPU
  - *access time to first byte*: time to lower level = \(f(\text{lower level latency})\)
  - *transfer time*: time to transfer block = \(f(\text{upper } \leftrightarrow \text{ lower data rate (BW)}, \text{ block size})\)

*Read block from memory*

*Remark*: servers care more about memory bandwidth because there is much more traffic to and from main memory due to many tasks and multiple processors. Battery-powered embedded systems care about average amount of energy (e.g., searches, bit transfers) per mission.
### Addressing

- **Address** = _Block frame address + block offset address_

<table>
<thead>
<tr>
<th>Address (memory pointer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block frame address</td>
</tr>
<tr>
<td>Block offset</td>
</tr>
</tbody>
</table>

- \( \# \text{bits(Block offset)} = \log_2(\text{block size}) \)

<table>
<thead>
<tr>
<th>Block</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Block size = 4 bytes
Offset = 2
Block Size vs. Cache Measures

- Increasing Block Size generally increases Miss Penalty and decreases Miss Rate

- Eventually: diminishing return; fewer independent blocks; larger transfer time
Implications For CPU

• Fast hit check is critical since done in every memory access
  – Hit is the common case

• Unpredictable memory access time
  – 10s of clock cycles: wait
  – 1000s of clock cycles:
    » Interrupt & switch & do something else

• How should we handle a miss?
  (10s of CPU cycles => HW, 1000s => SW)?
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)*

*Our focus is on entire blocks*
Q1: Where can a block be placed in the upper level? Associativity

**Set** – the block-size memory areas in which a given block of data may be placed

**Cache associativity (n):**
Set size / block size.

**Set # = Block frame address modulo number of sets**

When # of sets = \(2^n\), \(n>0\),

Set # = \((\log_2(\text{Number of Sets}))\)

Least significant bits of the Block frame address: the **Index bits**.

A set corresponds to a single horizontal plane
Contents of a Block

| v | Tag | D | D | ... | D |

- “Valid” bit: confirms that other bits are meaningful and not random junk.
- Tag: confirms that the data belongs to the correct block (Tag||Index = block address)
- D: actual “data” stored in the cache
Various Cache Organizations

Direct mapped = 1-way S.A.

2-way S.A.

4-way S.A.

Fully associative.
Q2: How Is a Block Found If It Is in the Upper Level?

- Index bits determine set
- Tag stored with each block serves to unambiguously confirm the block’s identity.
  - Compare address with all relevant tags in parallel + “valid” bit
  - (No need to check block offset (only required for specific byte))
  - (No need to check index (was used to choose the set))

- Increasing associativity with fixed cache size shrinks index (fewer sets), expands tag

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

Fully Associative: No index
Direct Mapped: Large index
Q3: Which Block Should be Replaced on a Miss?

- Direct Mapped does not need a replacement policy because there is no choice.

- Set Associative or Fully Associative:
  - Random (large associativities)
  - Least-Recently-Used (LRU) (smaller associativities)
  - First in – first out (FIFO)

Example: miss rates.

```
<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
```
Q4: What Happens on a Write Hit?

- **Write through**: The modified data is written to the block in the cache and to the block in the lower-level memory.

- **Write back**: The modified data is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced. Need a “dirty bit” in order to tag modified cache lines that need to be written back to the lower level when replaced.
Write Through vs. Write Back

- Both can be combined with write buffers so CPU does not need to wait for lower-level memory access to be completed.

- **WB:**
  - Less inter-level traffic on the bus in the case of repeated writes to block (good for servers and multi-processor machines)
  - Lower energy consumption (good for embedded systems)

- **WT:**
  - easier to implement
  - easier to maintain coherence (important for multi-processors and I/O)
  - Cache misses: no wait for writing to memory of the evacuated cache line because a copy already resides in lower level.

- Criterion: # of writes to a block during its lifetime in the cache.
Q4: What Happens on a Write Miss?

– **Write allocate:**
  » block is copied from lower level to cache
  » continue as if write hit

– **No write allocate (“write around”):**
  » block is not copied to cache
  » write only to the lower level (write around)

• **Common combinations:**
  – write back + write allocate (assumes that data will be accessed again soon)
  – write through + no write allocate (assumes no accesses in near future)
Example: 21064 Data Cache

- Index = 8 bits: 256 blocks = 8192/(32x1)

FIGURE 5.5 The organization of the data cache in the Alpha AXP 21064 microprocessor.
Write-Merging in Write Buffer

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**FIGURE 5.6** To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does.

- **Write buffer**: CPU writes words; each buffer entry must contain the full address, and has the width of a cache block; valid bits identify which word(s) are valid.
- **Problem**: poor utilization of buffer space
- **Solution**: when writing new word, check if relevant block number already has a buffer entry; if so, share it.

- **Benefits of Write buf**:
  - CPU doesn’t wait for copy to main memory (critical for WT cache)
- **With merging, also**:
  - reduced overhead
  - for same total buffer size, less likely that processor and cache will have to wait for buffer to empty

4 entry, 4 word

16 sequential writes in a row
2-way Set Associative, Address to Select Word

Two sets of Address tags and data RAM

Use address bits to select correct RAM

FIGURE 5.8 A two-way set-associative version of the 8-KB cache of Figure 5.5, showing the extra multiplexer in the path.
Separate vs. Unified Caches (Instruction And Data)

• Miss rate
  – Unified: flexible resource allocation
  – Separate: easier to tune to different access patterns

• The major problem - **structural conflict** in processor pipeline
  – Unified: same access path for both
  – Split: separate paths that can be used concurrently
## Separate vs. Unified Caches: Miss Rate

### Separate Instruction Cache and Data Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Instruction Cache</th>
<th>Data Cache</th>
<th>Unified Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>3.06%</td>
<td>24.61%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2 KB</td>
<td>2.26%</td>
<td>20.57%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4 KB</td>
<td>1.78%</td>
<td>15.94%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8 KB</td>
<td>1.10%</td>
<td>10.19%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16 KB</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32 KB</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64 KB</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.36%</td>
</tr>
<tr>
<td>128 KB</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>

**Fig. 5.7:** Relative weighting of instruction vs. data access must be taken into account!!!

Note: compare I+D with unified of next line (same total cache size)