Memory Hierarchies: Caches + Main Memory Organization

Today’s Menu:
- Basic Cache Functionality
- Basic Cache Design Parameters

CPU → MEM

Basic Cache Design
- Cache only holds a portion of a program
  - Which part of the program does the cache contain?
    - Cache holds most recently accessed references
    - Cache is divided into units called cache blocks (also known as cache “lines”), each block holds a contiguous set of memory addresses
    - How does the CPU know which part of the program the cache is holding?
      - Each cache block has extra bits, called the cache tag, which holds the main memory address of the data in the block

Cache Example
- Assume
  - r1 == 0, r2 == 1, r4 == 2
  - 1 cycle to cache
  - 6 cycles to main memory
  - Instructions take 1 cycle to execute

Small Cache Example (2)
- Assume
  - r1 == 0, r2 == 1, r4 == 2
  - 1 cycle to cache
  - 3 cycles to main memory
  - Instructions take 1 cycle to execute

Small Cache Example (3)
- Execute instruction add r1,r1,r2
  - PC == 0x00
  - Fetch instruction from memory system
    a) look into cache
    b) if not there, fetch from main memory and copy into cache
  - Tag says block 0 comes from address 0x00.
Small Cache Example (4)

At cycle 6:
- PC == 0x04
- Fetch instruction from memory system
  a) look into cache
  b) if not there, fetch from main memory and copy into cache
  → this takes 5 cycles

Small Cache Example (5)

At cycle 11:
- PC == 0x00
- Fetch instruction from memory system
  a) look into cache
  b) if not there, fetch from main memory
  → this takes 5 cycles

Small Cache Example (6)

At cycle 11:
- PC == 0x00
- Fetch instruction from memory system
  a) look into cache
  b) if not there, fetch from main memory
  → this takes 5 cycles

Small Cache Example (7)

At cycle 13:
- Execute bne r4,r1,0x00
  - Branch not taken

Small Cache Example (8)

At cycle 12:
- PC == 0x04
- Fetch instruction from memory system
  a) look into cache
  b) if not there, fetch from main memory
  → this takes 5 cycles

Small Cache Example (9)

At cycle 13:
- Execute bne r4,r1,0x00
  - Branch not taken
At cycle 18
- Put fetched instruction (sub r1,r1,r1) into cache
- Where in cache should we put it?
  (for now, randomly pick a block)

At cycle 17
- Put fetched instruction
- Where in cache should we put it?
  (for now, randomly pick a block)

At cycle 13
- Put fetched instruction (sub r1,r1,r1) into cache
- Where in cache should we put it?
  (for now, randomly pick a block)
The ABC's (or 1-2-3-4's) of Caches

- Caching is a general concept used in processors, operating systems, file systems, and applications.
- Wherever it is used, there are four basic questions which arise.
  - Q1: Where can a block be placed in a cache? (block placement)
  - Q2: How is a block found if it is in a cache? (block identification)
  - Q3: Which block should be replaced on a miss? (block replacement)
  - Q4: What happens on a write? (write strategy)

Q1: Block Placement

**NO CACHE**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Address</th>
<th>Op/Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>6-10</td>
<td>0x04</td>
<td>FETCH 0x04</td>
</tr>
<tr>
<td>11-15</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>16-20</td>
<td>0x04</td>
<td>FETCH 0x04</td>
</tr>
<tr>
<td>21-25</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>26-30</td>
<td>0x0C</td>
<td>j, 0x00</td>
</tr>
</tbody>
</table>

Different approaches to block placement:
- Direct Mapped: block 4 goes only into block 4 (12 mod 8)
- Fully associative: block 12 can go anywhere (12 mod 8)

Q1: Block Placement

**CACHE**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Address</th>
<th>Op/Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>6-10</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>11-15</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>16-20</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>21-25</td>
<td>0x00</td>
<td>FETCH 0x00</td>
</tr>
<tr>
<td>26-30</td>
<td>0x0C</td>
<td>j, 0x00</td>
</tr>
</tbody>
</table>

Each memory location maps to one cache line. Mapping is done by a modulo operator, which is accomplished by ignoring some of the MSBs (e.g., address 001100 = 12 and 000100 = 4 both get mapped to 100 = 4)

Q1: Block Placement

- Every memory location maps to exactly one set of cache lines. Within a set it can go in any location.
- Tradeoff between simplicity (direct-mapped to sets) and flexibility (fully associative within sets).

Q1: Block Placement

- Any memory location can go to any cache line. This is infinitely flexible, but... it requires very complex (and slow) hardware.

Q4: Write Strategy

- Write-back: Cache stores a copy of the memory block.
- Write-through: Cache does not store a copy of the memory block.
- Write-through with dirty bits: Cache tracks the status of each block.

Q4: Write Strategy

- Write-back: Cache stores a copy of the memory block.
- Write-through: Cache does not store a copy of the memory block.
- Write-through with dirty bits: Cache tracks the status of each block.
Q2: Block Identification

- Every block has an address tag that stores the main memory address of the data stored in the block.
- When checking the cache, the processor will compare the requested memory address to the cache tag — if the two are equal, then there is a cache hit and the data is present in the cache.
- Often, each cache block also has a valid bit that tells if the contents of the cache block are valid.

Q3: Block Replacement

- Several different replacement policies can be used:
  - Random replacement - randomly pick any block
  - Easy to implement in hardware, just requires a random number generator
  - Spreads allocation uniformly across cache
  - May evict a block that is about to be accessed
  - Least-recently used (LRU) - pick the block in the set which was least recently accessed
  - Assumed more recently accessed blocks more likely to be referenced again
  - Easy to implement for 2-way set-associative caches
  - Just have a single bit which gets set when the block is accessed, unset when the other block in the set is accessed
  - Much more difficult to implement for set-associativity greater than 2
  - Must resort to pseudo LRU
  - One bit specifies most recently used block and replacement algorithm randomly picks from the remaining blocks in the set

Q4: Write Strategy

- When data is written into the cache (on a store), is the data also written to main memory?
  - If data is written to memory, the cache is called a write-through cache
  - PRO: Easy hardware
  - PRO: Good speed on read misses
  - If data is NOT written to memory, the cache is called a write-back cache
  - For a write-back cache, need to remember which blocks have been written ("dirty bit")
  - When you replace a "dirty" block, it must first be written back into the main memory
  - PRO: Good for "write-intensive" apps
**Q4: Write Strategy (Write Allocation)**

- Should you cache a block if it is a write-miss?
  - Maybe it will not be read again
  - Maybe it will replace a block that could be read again

- Write allocate:
  - On ANY miss (write or read) move the missed block into the cache

- Write no-allocate:
  - Only move missed block into the cache in case of read-miss

**Impact of Cache Size**

**Miss Ratio:** % access to cache that miss

![Graph showing miss ratio as a function of cache size. The graph labeled StatStack-min and StatStack-max shows the minimum and maximum of memory accesses in the address trace.](image)

**Better Metric: Average Memory Access time**

- Average time to access memory (AMAT)

  Average memory access time = Hit time + (Miss Rate \times Miss Penalty)

  where miss penalty is the **extra** time it takes to handle a miss (i.e., the time beyond the required 1 cycle hit cost)

  ![Diagram showing AMAT calculation](image)

**Summary of Cache Questions**

- Q1: Where can a block be placed in the cache?
  - Direct mapped, associative, fully-associative [Block Placement]

- Q2: How is a block found if it is in the cache?
  - indexing (direct mapped), limited search (associative), full search (fully-associative) [Block Identification]

- Q3: Which block should be replaced on a cache miss?
  - random, least-recently used (LRU) [Block Replacement]

- Q4: What happens on a write?
  - write-through or write-back [Write Strategy]

**Metric for Cache Performance - Miss Rate (SPEC 92)**

- 32 Byte block size, direct-mapped caches
  - Option 1: 2 separate caches, one for instructions, one data; a Split cache
  - Option 2: A single cache with both instructions & data; a Unified cache

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Instruction Cache</th>
<th>Data Cache</th>
<th>Unified Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>3.06%</td>
<td>24.61%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2KB</td>
<td>2.26%</td>
<td>20.57%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4KB</td>
<td>1.78%</td>
<td>15.94%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8KB</td>
<td>1.10%</td>
<td>10.19%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16KB</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32KB</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64KB</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.35%</td>
</tr>
<tr>
<td>128KB</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>

**Average Memory Access Time Example**

- Recall:
  - Average memory access time = Hit time + (Miss Rate \times Miss Penalty)

- Example 1:
  - Machine has 1 cycle hit cost, 10 cycle miss penalty (11 cycles total for a miss)
  - Program has 10% miss rate
  - Average memory access time = 1.0 + 10\% \times 10 = 2.0

- Example 2:
  - Same machine as example 1
  - Program has 100 memory references; 90 hit, 10 miss
  - 90 hits \times 1 cycle + 10 misses \times 11 cycles = 200 cycles
  - Average memory access time = \# of cycles / \# of refs = 200 cycles / 100 refs = 2.0
Example

> Which system has a lower miss rate?
  - 16KB instruction cache & 16 KB data cache
  - 32KB unified instruction+data cache

- Assume a hit takes 1 cycle, a miss costs 50 cycles and a load or store takes 1 extra clock cycle on the unified cache.
- Why the extra cycle?
  - Because the unified cache only has one port to satisfy simultaneous requests.
- Assume that 75% of the memory references are instruction fetches.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Instruction Cache</th>
<th>Data Cache</th>
<th>Unified Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>3.06%</td>
<td>24.61%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2KB</td>
<td>2.26%</td>
<td>20.57%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4KB</td>
<td>1.78%</td>
<td>15.94%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8KB</td>
<td>1.19%</td>
<td>10.19%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16KB</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32KB</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64KB</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.39%</td>
</tr>
<tr>
<td>128KB</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>

Example (cont.)

> Solution
  - Using the miss rates from the SPEC92 benchmark suite.
    - For the split caches, the miss rate is:
      \[(75\% \times 0.64\%) + (25\% \times 6.47\%) = 2.10\%
    - For the unified cache, the miss rate is (taken directly from the table): 1.99%.
    - Given result, a designer optimizing miss rate would build a 32KB unified cache.

> But, if we consider the cycle time penalty for a unified cache, we can compute the average memory access time:

\[
\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Memory stall cycles/Instruction}) \times \text{Clock cycle time}
\]

Where: IC = Instruction Count.

Example

> What’s the impact on performance (CPU time) when the following cache behavior is included:
  - 50 cycle miss penalty.
  - All instructions normally take 2.0 cycles (excluding memory stalls).
  - Miss rate is 2.0%.
  - Average of 1.33 memory references per instruction.

Recall that:

\[
\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Memory stall cycles/Instruction}) \times \text{Clock cycle time}
\]

Example (cont.)

> What's the impact on performance (CPU time) when the following cache behavior is included:
  - 50 cycle miss penalty.
  - All instructions normally take 2.0 cycles (excluding memory stalls).
  - Miss rate is 2.0%.
  - Average of 1.33 memory references per instruction.

Recall that:

\[
\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{exec}} + \text{Memory stall cycles/Instruction}) \times \text{Clock cycle time}
\]

Where: IC = Instruction Count.
Performance Impact of Cache

- How much does cache impact application performance?
  - As its cache size is decreased its CPI increases by

Real Life Examples: Pentium Pro

Real Life Examples: Alpha DEC

Real Life Examples: PowerPC

Summary: Levels of the Memory Hierarchy