Memory Hierarchy: Cache

Memory hierarchy
Cache basics
Locality
Cache organization
Cache-aware programming

Cache Hit

1. Request data in block \( b \).
2. **Cache hit**: Block \( b \) is in cache.

Cache Miss

1. Request data in block \( b \).
2. **Cache miss**: block is not in cache
3. **Cache eviction**: Evict a block to make room, maybe store to memory.
4. **Cache fill**: Fetch block from memory, store in cache.

General Cache Mechanics

Block: unit of data in cache and memory. (a.k.a. line)
Smaller, faster, more expensive. Stores subset of memory blocks. (lines)

Larger, slower, cheaper. Partitioned into blocks (lines).

CPU

Cache

Data is moved in block units

Memory

Placement Policy: where to put block in cache
Replacement Policy: which block to evict
### Locality #1

**Data:**

Temporal: referenced in each iteration

Spatial: array `a[]` accessed in stride-1 pattern

**Instructions:**

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

A assessing locality in code is an important programming skill.

```c
int sum = 0;
for (i = 0; i < n; i++) {
    sum += a[i];
}
return sum;
```

What is stored in memory?

### Locality #2

**Data:**

Temporal: `sum` is accessed in each iteration

Spatial: `a[]` is accessed in stride-1 pattern

**Instructions:**

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

```c
int sum_array_rows(int a[M][N]) {
    int sum = 0;
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

**Data:**

Temporal: `sum` and `a[][]` are accessed in each iteration

Spatial: `a[][]` is accessed in row-major pattern

**Instructions:**

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

```c
int sum_array_cols(int a[M][N]) {
    int sum = 0;
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < M; i++) {
            sum += a[i][j];
        }
    }
    return sum;
}
```

What is "wrong" with this code?
How can it be fixed?

### Locality #3

**Data:**

Temporal: `sum` and `a[][]` are accessed in each iteration

Spatial: `a[][]` is accessed in row-major pattern

**Instructions:**

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

```c
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

### Locality #4

**Data:**

Temporal: `sum` and `a[][][]` are accessed in each iteration

Spatial: `a[][][]` is accessed in row-major pattern

**Instructions:**

Temporal: execute loop repeatedly

Spatial: execute instructions in sequence

```c
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

What is "wrong" with this code?
How can it be fixed?
Cache Performance Metrics

**Miss Rate**
Fraction of memory accesses to data not in cache (misses / accesses)
Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

**Hit Time**
Time to find and deliver a block in the cache to the processor.
Typically: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

**Miss Penalty**
Additional time required on cache miss = main memory access time
Typically 50 - 200 cycles for L2 (trend: increasing!)

Cache Organization: Key Points

**Block**
Fixed-size unit of data in memory/cache

**Placement Policy**
Where in the cache should a given block be stored?
- direct-mapped, set associative

**Replacement Policy**
What if there is no room in the cache for requested data?
- least recently used, most recently used

**Write Policy**
When should writes update lower levels of memory hierarchy?
- write back, write through, write allocate, no write allocate

memory hierarchy
why does it work?

small, fast, power-hungry, expensive

L1 cache
(SRAM, on-chip)

L2 cache
(SRAM, on-chip)

L3 cache
(SRAM, off-chip)

main memory
(DRAM)

persistent storage
(hard disk, flash, over network, cloud, etc.)

Blocks
Divide address space into fixed-size aligned blocks. power of 2

Example: block size = 8

full byte address

Block ID: 00010010

offset within block

log₂(block size)

(address bits - offset bits)

Note: drawing address order differently from here on!

remember withinSameBlock? (Pointers Lab)
Placement: *Direct-Mapped*

**Mapping:**
\[ \text{index} (\text{Block ID}) = \text{Block ID} \mod S \]

(easy for power-of-2 block sizes...)

---

**Placement:** Tags resolve ambiguity

**Mapping:**
\[ \text{index} (\text{Block ID}) = \text{Block ID} \mod S \]

---

**Address = Tag, Index, Offset**

Block ID bits - Index bits \( \log_2(\# \text{cache slots}) \)

Tag \quad Index \quad Offset

\( (a-b) \) bits \quad s \text{ bits} \quad b \text{ bits}

---

**A puzzle.**

Cache starts empty.

Access (address, hit/miss) stream:

(10, miss), (11, hit), (12, miss)

What could the block size be?
Placement: direct mapping conflicts

What happens when accessing in repeated pattern: 0010, 0110, 0010, 0110, 0010...?

cache conflict
Every access suffers a miss, evicts cache line needed by next access.

Example: Tag, Index, Offset?

4-bit Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

Direct-mapped
tag bits _____ 4 slots 2-byte blocks

index(1101) = _____

Example: Tag, Index, Offset?

E-way set-associative
S slots 16-byte blocks

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

E = 1-way S = 8 sets

E = 2-way S = 4 sets

E = 4-way S = 2 sets

Placement: Set Associative

One index per set of block slots. Store block in any slot within set.

Mapping:
index(Block ID) = Block ID mod S

Replacement policy: if set is full, what block should be replaced?
Common: least recently used (LRU) but hardware usually implements “not most recently used”
Replacement Policy

If set is full, what block should be replaced?

Common: least recently used (LRU)
(but hardware usually implements “not most recently used”)

Another puzzle: Cache starts empty, uses LRU.
Access (address, hit/miss) stream

(10, miss); (12, miss); (10, miss)

associativity of cache?

General Cache Organization (S, E, B)

Powers of 2

E lines per set (“E-way”)

S sets

cache capacity: $S \times E \times B$ data bytes
address size: $t + s + b$ address bits

Direct-Mapped Cache Practice

12-bit address
16 lines, 4-byte block size

Direct mapped

Offset bits? Index bits? Tag bits?

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>60</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>38</td>
</tr>
<tr>
<td>2</td>
<td>08</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>18</td>
<td>D3</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

0x354
0xA20

0x354
0xA20
Example (E = 1)

Locals in registers.
Assume a is aligned such that
\( a[r][c] \) & aa...arr rrc ccc000

int sum_array_rows(double a[16][16]){
    double sum = 0;
    for (int r = 0; r < 16; r++){
        for (int c = 0; c < 16; c++){
            sum += a[r][c];
        }
    }
    return sum;
}

int sum_array_cols(double a[16][16]){
    double sum = 0;
    for (int c = 0; c < 16; c++){
        for (int r = 0; r < 16; r++){
            sum += a[r][c];
        }
    }
    return sum;
}

Assume: cold (empty) cache
3-bit set index, 5-bit offset
\( 0:0:a...a000 \) 000 00000

Example (E = 1)

int dotprod(int x[8], int y[8]) {
    float sum = 0;
    for (int i = 0; i < 8; i++) {
        sum += x[i]*y[i];
    }
    return sum;
}

if x and y are mutually aligned, e.g., 0x00, 0x80

Addresses as bits
0x00000000:
0x00000080:
0x000000A0:

Writing to cache

Multiple copies of data exist, must be kept in sync.

Write-hit policy
  Write-through:
  Write-back: needs a dirty bit

Write-miss policy
  Write-allocate:
  No-write-allocate:

Typical caches:
  Write-back + Write-allocate, usually
  Write-through + No-write-allocate, occasionally
Write-back, write-allocate example

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>0xCAFÉ</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty Bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>T</th>
<th>0xFACE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0xCAFÉ</td>
<td></td>
</tr>
</tbody>
</table>

1. mov $T, %ecx
2. mov $U, %edx
3. mov $0xFEED, (%ecx)
   a. Miss on T.

Write-back, write-allocate example

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>0xFEED</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty Bit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>T</th>
<th>0xFACE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0xCAFÉ</td>
<td></td>
</tr>
</tbody>
</table>

1. mov $T, %ecx
2. mov $U, %edx
3. mov $0xFEED, (%ecx)
   a. Miss on T.
   c. Fill T (write-allocate).
   d. Write T in cache (dirty).
4. mov (%edx), %eax
   a. Miss on U.
   b. Evict T (dirty: write back).
   c. Fill U.
   d. Set %eax.
5. DONE.