Agenda

• Caching
General Cache Mechanics

Cache

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Memory

Larger, slower, cheaper memory viewed as partitioned into “blocks”
General Cache Concepts: Hit

Cache

Request: 14

Memory
General Cache Concepts: Hit

Data in block 14 is needed

Cache

Request: 14

8  9  14  3

Memory

0  1  2  3
4  5  6  7
8  9  10 11
12 13 14 15
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

Data in block b is needed

Request: 12

Cache

8  9  14  3

Memory

0  1  2  3
4  5  6  7
8  9  10 11
12 13 14 15
General Cache Concepts: Miss

Request: 12

Data in block b is needed

Block b is not in cache: Miss!
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory
**General Cache Concepts: Miss**

Data in block b is needed

Block b is not in cache:
**Miss!**

Block b is fetched from memory

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
Cache Performance Metrics

**Miss Rate**
- Fraction of memory references not found in cache (misses / accesses)
  - $= 1 - \text{hit rate}$
- Typical numbers (in percentages):
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

**Hit Time**
- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 1-2 clock cycle for L1
  - 5-20 clock cycles for L2
  - 30-50 clock cycles for L3

**Miss Penalty**
- Additional time required because of a miss
  - typically 100-400 cycles for main memory (trend: increasing!)
Lets think about those numbers

■ Huge difference between a hit and a miss
  ▪ Could be 100x, if just L1 and main memory

■ Would you believe 99% hits is twice as good as 97%?
  ▪ Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  ▪ Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

■ This is why “miss rate” is used instead of “hit rate”
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block $i$ must be placed in slot $i \mod \text{size}$), *direct-mapped*
    - if more than one, $n$-way *set-associative* (where $n$ is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache (just won’t fit)
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality:**
  - Recently referenced items are *likely* to be referenced again in the near future.

- **Spatial locality:**
  - Items with nearby addresses *tend* to be referenced close together in time.

- How do caches take advantage of this?
Example: Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**
General Cache Organization (S, E, B)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
- \( B = 2^b \) bytes data block per cache line (the data)

Cache size: \( S \times E \times B \) data bytes
Cache Read

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

S = 2^s sets
E = 2^e lines per set

Address of word:
- t bits
- s bits
- b bits

- tag
- set
- index
- block
- offset

B = 2^b bytes data block per cache line (the data)
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

\[ S = 2^s \text{ sets} \]

Address of int:
\[
\begin{array}{c}
\text{t bits} \\
0...01 \\
100
\end{array}
\]

find set
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

valid? + match: assume yes = hit

block offset
Example: Direct-Mapped Cache (E = 1)

Direct-mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
Example (for $E = 1$)

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Assume sum, i, j in registers
Address of an aligned element of $a$: $aa....aaaxxxxyyyy000$

Assume: cold (empty) cache
3 bits for set, 5 bits for byte
$aa....aaaaxx xyy yyy000$

32 B = 4 doubles
32 B = 4 doubles
Example (for $E = 1$)

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

If $x$ and $y$ have aligned starting addresses, e.g., $&x[0] = 0$, $&y[0] = 128$

If $x$ and $y$ have unaligned starting addresses, e.g., $&x[0] = 0$, $&y[0] = 144$
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

find set

- t bits
- 0...01
- 100
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

t bits 0...01 100

valid? + match: yes = hit

compare both

block offset

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

t bits 0...01 100

valid? + match: yes = hit

compare both

block offset
E-way Set-Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

Address of short int: t bits 0...1 100

match both

valid? + match: yes = hit

short int (2 Bytes) is here

block offset
Example (for E = 2)

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

if x and y have aligned starting addresses, e.g., &x[0] = 0, &y[0] = 128 still can fit both because 2 lines in each set
Fully Set-Associative Caches (S = 1)

- All lines in one single set, S = 1
  - E = C / B, where C is total cache size
  - S = 1 = ( C / B ) / E

- Direct-mapped caches have E = 1
  - S = ( C / B ) / E = C / B

- Tags are more expensive in associative caches
  - Fully-associative cache, C / B tag comparators
  - Direct-mapped cache, 1 tag comparator
  - In general, E-way set-associative caches, E tag comparators

- Tag size, assuming m address bits (m = 32 for IA32)
  - m – log₂S – log₂B
What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate