Types of Cache Misses

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

- **Conflict miss**
  - Occurs when some block is evicted out of the cache, but then that block is referenced again later
  - Conflict misses occur when the cache is large enough, but multiple data blocks all map to the same slot
    - e.g., if blocks 0 and 8 map to the same cache slot, then referencing 0, 8, 0, 8, ... would miss every time
    - Conflict misses may be reduced by increasing the **associativity** of the cache

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

**General Cache Organization (S, E, B)**

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

**Cache Read**

- $E = 2^e$ lines per set
- $S = 2^s$ sets
- Address of byte in memory:
  - $t$ bits tag
  - $s$ bits set index
  - $b$ bits block offset
  - Data begins at this offset

- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset
**Example: Direct-Mapped Cache (E = 1)**

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

<table>
<thead>
<tr>
<th>true</th>
<th>tag</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>
</tr>
</thead>
</table>

S = 2^E sets

<table>
<thead>
<tr>
<th>true</th>
<th>tag</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>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>true</th>
<th>tag</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>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>true</th>
<th>tag</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>
</tr>
</thead>
</table>

**Address of int:**

- 1 bits: 0...01 100

### Example (for E = 1)

The code snippet defines two functions: `sum_array_rows` and `sum_array_cols` to calculate the sum of all elements in a 2D array and each row or column, respectively.

```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;
}
```

These functions are used to compute the sum of elements in a given array. The example code snippet demonstrates how to implement these calculations efficiently.

- **Example:**
  - **Direct-Mapped Cache (E = 1)**
  - **Address of int:**
    - 1 bits: 0...01 100
  - **Find set:**
  - **Block offset:**
  - **Valid? + Match?: yes = hit**
  - **No match:** old line is evicted and replaced

- **Assume:**
  - Sum, i, j in registers
  - Address of an aligned element of `a`: aa...ayyyyyxxxx000

- **Assume:**
  - Cold (empty) cache
  - 3 bits for set, 5 bits for offset
  - 32 B = 4 doubles

- **4 misses per row of array 4^16 = 64 misses**
- **32 B = 4 doubles**
- **Every access a miss 16^16 = 256 misses**
Example (for E = 1)

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

In this example, cache blocks are 16 bytes; 8 sets in cache

How many block offset bits?

How many set index bits?

Address bits: ttt...ttssbbbb

B = 8 = 2^3

s = 3

offset bits

index bits

0:

128:

160:

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] = 160

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

E = 2: Two lines per set

Assume: cache block size 8 bytes

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),...
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. $&x[0] = 0$, $&y[0] = 128$, can still fit both because two lines in each set.

Fully Set-Associative Caches ($S = 1$)

- Fully-associative caches have all lines in one single set, $S = 1$
  - $E = C / B$, where $C$ is total cache size
  - Since, $S = (C / B) / E$, therefore, $S = 1$
- Direct-mapped caches have $E = 1$
  - $S = (C / B) / E = C / B$
- Tag matching is more expensive in associative caches
  - Fully-associative cache needs $C / B$ tag comparators: one for every line!
  - Direct-mapped cache needs just 1 tag comparator
  - In general, an $E$-way set-associative cache needs $E$ tag comparators
- Tag size, assuming $m$ address bits ($m = 32$ for IA32):
  - $m - \log_2S - \log_2B$

Intel Core i7 Cache Hierarchy

Processor package

Core 0
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

L1 i-cache and d-cache: 32 KB, 8-way, Access: 4 cycles
L2 unified cache: 256 KB, 8-way, Access: 11 cycles
L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles
Block size: 64 bytes for all caches.

What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until line is evicted)
    - Need a dirty bit to indicate if line is 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 (just write immediately to memory)
- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Software Caches are More Flexible

- **Examples**
  - File system buffer caches, web browser caches, etc.

- **Some design differences**
  - Almost always fully-associative
    - so, no placement restrictions
    - index structures like hash tables are common (for placement)
  - Often use complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single “block” transfers
    - may fetch or write-back in larger units, opportunistically

Optimizations for the Memory Hierarchy

- **Write code that has locality**
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- **How to achieve?**
  - Proper choice of algorithm
  - Loop transformations

---

**Example: Matrix Multiplication**

```c
void mmn(double *a, double *b, double *c, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
```

- Assume:
  - Matrix elements are doubles
  - Cache block = 64 bytes = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 + n = 9n/8 misses (omitting matrix c)

- **Afterwards in cache:**
  - (schematic)
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 64 bytes = 8 doubles
  - Cache size $C << n$ (much smaller than $n$)

- Other iterations:
  - Again:
    - $n/8 \times n = 9n/8$ misses (omitting matrix $c$)

- Total misses:
  - $9n/8 \times n^2 = (9/8) \times n^3$

Blocked Matrix Multiplication

```c
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

Cache Miss Analysis

- Assume:
  - Cache block = 64 bytes = 8 doubles
  - Cache size $C << n$ (much smaller than $n$)
  - Three blocks fit into cache: $3B^2 < C$

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix $c$)

- Afterwards in cache (schematic)

- Other (block) iterations:
  - Same as first iteration
  - $2n/B \times B^2/8 = nB/4$

- Total misses:
  - $nB/4 \times (nB)^2 = n^3/(4B)$
Summary

- No blocking: \((9/8) \times n^3\)
- Blocking: \(1/(4B) \times n^3\)
- If \(B = 8\) difference is \(4 \times 8 \times 9 / 8 = 36x\)
- If \(B = 16\) difference is \(4 \times 16 \times 9 / 8 = 72x\)

- Suggests largest possible block size \(B\), but limit \(3B^2 < C\!\!\!\!\)\(^2\)

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: \(3n^2\), computation \(2n^3\)
    - Every array element used \(O(n)\) times!
  - But program has to be written properly

Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip