Cache Memory Organization and Access
Example Memory Hierarchy

- **L0:** CPU registers hold words retrieved from the L1 cache.
- **L1:** L1 cache holds cache lines retrieved from the L2 cache.
- **L2:** L2 cache holds cache lines retrieved from L3 cache.
- **L3:** L3 cache holds cache lines retrieved from main memory.
- **L4:** Main memory holds disk blocks retrieved from local disks.
- **L5:** Local disks hold files retrieved from disks on remote servers.
- **L6:** Remote secondary storage (e.g., Web servers).

- Smaller, faster, and costlier (per byte) storage devices
- Larger, slower, and cheaper (per byte) storage devices
General Cache Concept

Cache

| 4 | 9 | 10 | 3 |

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

Data is copied in block-sized transfer units

Memory

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

Larger, slower, cheaper memory viewed as partitioned into “blocks”
Cache Memories

- **Cache memories** are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:
General Cache Organization (S, E, B)

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

Example diagram showing set, line, valid bit, and tag.
Cache Read

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

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

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

Data begins at this offset
Example: Direct Mapped Cache (E = 1)

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

S = 2^s sets

Address of int:

\[\text{t bits} \quad 0\ldots01 \quad 100\]
Example: Direct Mapped Cache ($E = 1$)

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

[Diagram of direct-mapped cache with addresses and offsets]

Address of int: 
```
  t bits 0...01 100
```

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

If tag doesn’t match: old line is evicted and replaced
Example: Direct-Mapped Cache Simulation

M=16 bytes (4-bit addresses), B=2 bytes/block, S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Ta</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

t=1  s=2  b=1
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 |
```

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

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

Address of short int:

compare both

match: yes = hit

valid? +

block offset
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

Address of short int:

block offset

short int (2 Bytes) is here

No match:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), …
Example: 2-Way Set Associative Cache Simulation

M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0  [0000₂], miss
1  [0001₂], hit
7  [0111₂], miss
8  [1000₂], miss
0  [0000₂]  hit

<table>
<thead>
<tr>
<th>v</th>
<th>Ta</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

Set 0

Set 1

\(t=2\) s=1 \(b=1\)
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 straight to memory, does not load into cache)

- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate
Intel Core i7 Cache Hierarchy

Processor package

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

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

L1 i-cache and d-cache: 
32 KB, 8-way, 
Access: 4 cycles

L2 unified cache: 
256 KB, 8-way, 
Access: 10 cycles

L3 unified cache: 
8 MB, 16-way, 
Access: 40-75 cycles

Block size: 64 bytes for all caches.
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:
    - 4 clock cycle for L1
    - 10 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)
Let’s 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”**
Writing Cache Friendly Code

- Make the **common** case go fast
  - Focus on the **inner loops** of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (**temporal locality**) because there is a good chance that they are stored in registers.
  - Stride-1 reference patterns are good (**spatial locality**) because subsequent references to elements in the same block will be able to hit the cache (one cache miss followed by many cache hits).
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain:** Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.
Rearranging Loops to Improve Spatial Locality
Matrix Multiplication Example

Description:

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations
- N reads per source element
- N values summed per destination
  - but may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Miss Rate Analysis for Matrix Multiply

Assume:
- Block size = 32B (big enough for four doubles)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:
- Look at access pattern of inner loop
## Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations

- **Stepping through columns in one row:**
  ```c
  for (i = 0; i < N; i++)
      sum += a[0][i];
  ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B

- **Stepping through rows in one column:**
  ```c
  for (i = 0; i < n; i++)
      sum += a[i][0];
  ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)
Matrix Multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Inner loop:

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jik)

/* jik */
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum
    }
}

Inner loop:

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.25</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

**Inner loop:**

- **(i,k)** (Fixed)
- **(k,*)** (Row-wise)
- **(i,*)** (Row-wise)

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (ijk)

```c
/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Matrix Multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Inner loop:

A: Column-wise

B: Fixed

C: Column-wise
Matrix Multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += a[i][k] * r;
    }
}
```

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

ijk (& jik):
• 2 loads, 0 stores
• misses/iter = 1.25

for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}

for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = a[i][k];
        for (j=0; j<n; j++)
            c[i][j] += r * b[k][j];
    }
}

for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += r * b[k][j];
    }
}

kij (& ikj):
• 2 loads, 1 store
• misses/iter = 0.5

for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = b[k][j];
        for (i=0; i<n; i++)
            c[i][j] += r * b[k][j];
    }
}

jki (& kji):
• 2 loads, 1 store
• misses/iter = 2.0
Core i7 Matrix Multiply Performance

![Graph showing Core i7 Matrix Multiply Performance](image)

- **ji / ki**
- **ijk / jik**
- **kij / ikj**
Cache Summary

- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.