Cache Memory Organization and Access
Example Memory Hierarchy

- **Reg**s
- L1 cache (SRAM)
- L2 cache (SRAM)
- **L3** cache (SRAM)
- **L4**: Main memory (DRAM)
- **L5**: Local secondary storage (local disks)
- **L6**: Remote secondary storage (e.g., Web servers)

Smaller, faster, and costlier (per byte) storage devices:
- L0: CPU registers hold words retrieved from the L1 cache.
- L1 cache holds cache lines retrieved from the L2 cache.
- L2 cache holds cache lines retrieved from L3 cache.
- 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.
General Cache Concept

Cache

0  1  2  3
4  9 10  3

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

Data is copied in block-sized transfer units

Memory

0  1  2  3
4  5  6  7
8  9 10 11
12 13 14 15

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)

\[ E = 2^e \text{ lines per set} \]

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

\[ B = 2^b \text{ bytes per cache block (the data)} \]

Cache size:
\[ C = S \times E \times B \text{ data bytes} \]
Cache Read

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

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

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

- **tag**
- **set index**
- **block offset**

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

**Direct mapped: one line per set**

Assume: cache block size 8 bytes

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
v & \text{tag} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\end{array}
\]

Address of int:

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{t bits} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\]

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

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

valid? + match: assume yes = hit

Address of int:

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>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>miss</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td>1</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>

\[s=2 \quad b=1\]
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:

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), …
Example: 2-Way Set Associative Cache Simulation

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

Address trace (reads, one byte per read):

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

```plaintext
<table>
<thead>
<tr>
<th>t=2</th>
<th>s=1</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
```
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 Example

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.

Main memory
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be very 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

Ouch!
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: $0.97 \times 1 \text{ cycle} + 0.03 \times 100 \text{ cycles} \approx 1 \text{ cycle} + 3 \text{ cycles} = 4 \text{ cycles}$
      - 99% hits: $0.99 \times 1 \text{ cycle} + 0.01 \times 100 \text{ cycles} \approx 1 \text{ cycle} + 1 \text{ cycle} = 2 \text{ cycles}$

- **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).
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:
  
  ```
  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:
  
  ```
  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;
    }
}
```

We can reorganize the loops in several different ways. Which organization gives the best cache performance?
Matrix Multiplication

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 (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];
}

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] += a[i][k] * r;
}

ijk (& jik)

kij (& ikj)

jki (& kji)
Measuring Cache Misses

valgrind with cachegrind - cache simulator

<table>
<thead>
<tr>
<th>Dlmr</th>
<th>Dlmw</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,977,062,519</td>
<td>0</td>
<td>ijk</td>
</tr>
<tr>
<td>4,004,000,016</td>
<td>0</td>
<td>jki</td>
</tr>
<tr>
<td>1,253,500,008</td>
<td>0</td>
<td>jik</td>
</tr>
<tr>
<td>750,750,003</td>
<td>0</td>
<td>kji</td>
</tr>
<tr>
<td>252,004,004</td>
<td>12</td>
<td>ikj</td>
</tr>
<tr>
<td>252,004,004</td>
<td>12</td>
<td>kij</td>
</tr>
</tbody>
</table>

L1 cache = 1024B
16 way associative
32 B cache line

<table>
<thead>
<tr>
<th>Dlmr</th>
<th>Dlmw</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,133,750,020</td>
<td>0</td>
<td>ijk</td>
</tr>
<tr>
<td>5,005,000,020</td>
<td>0</td>
<td>jki</td>
</tr>
<tr>
<td>5,005,000,020</td>
<td>0</td>
<td>kji</td>
</tr>
<tr>
<td>2,820,375,018</td>
<td>0</td>
<td>jik</td>
</tr>
<tr>
<td>189,003,003</td>
<td>9</td>
<td>ikj</td>
</tr>
<tr>
<td>189,003,003</td>
<td>9</td>
<td>kij</td>
</tr>
</tbody>
</table>

L1 cache = 1024B
4 way associative
32 B cache line
Core i7 Matrix Multiply Performance

- $ijk$ / $jik$
- $jki$ / $kji$
- $ijk$ / $jik$
- $kij$ / $ikj$
Learn about your machine's cache

- **lshw command** - list hardware information
  - `sudo lshw -C memory`
- **lscpu command** - display information about the CPU architecture
  - `lscpu`
- **dmidecode command** -
  - `sudo dmidecode -t cache`

Note: some of these may not work well in a virtual machine environment.
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.