Cache Memories

Today

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

- **Reg**: CPU registers hold words retrieved from the L1 cache.
- **L1 cache (SRAM)**: L1 cache holds cache lines retrieved from the L2 cache.
- **L2 cache (SRAM)**: L2 cache holds cache lines retrieved from L3 cache.
- **L3 cache (SRAM)**: L3 cache holds cache lines retrieved from main memory.
- **Main memory (DRAM)**: Main memory holds disk blocks retrieved from local disks.
- **Local secondary storage (local disks)**: Local disks hold files retrieved from disks on remote servers.
- **Remote secondary storage (e.g., Web servers)**: Larger, slower, and cheaper (per byte) storage devices.
General Cache Concept

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”
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:**

![Diagram of a typical system structure showing the integration of cache memory, CPU chip, register file, ALU, bus interface, I/O bridge, and main memory.](Image)
General Cache Organization \((S, E, B)\)

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

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

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

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

- 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

\[ B = 2^b \text{ bytes per cache block (the data)} \]
Example: Direct Mapped Cache (E = 1)

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

Address of int:

S = 2^s sets

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
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):
0  [0000_2],   miss
1  [0001_2],   hit
7  [0111_2],   miss
8  [1000_2],   miss
0  [0000_2]    miss

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</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>0</td>
</tr>
</tbody>
</table>
E-way Set Associative Cache (Here: \( E = 2 \))

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

Address of short int:

\[
\begin{array}{c|c|c}
\text{t bits} & 0...01 & 100 \\
\end{array}
\]

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

valid? + match: yes = hit

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:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...01</th>
<th>100</th>
</tr>
</thead>
</table>

valid? + match: yes = hit

compare both

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), ...
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):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Block</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[000002],</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[00012],</td>
<td>hit</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[01112],</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[10002],</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[00002]</td>
<td>hit</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></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

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

L3 unified cache (shared by all cores)

Main memory

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 – 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)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories
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.
```c
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of array “data” with stride of "stride", using using 4x4 loop unrolling. */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return (acc0 + acc1) + (acc2 + acc3);
}
```

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput (MB/s)
The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

Read throughput (MB/s) vs Size (bytes) vs Stride (x8 bytes)
Writing Cache-Friendly Code

- 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.