Memory Hierarchy

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Example Memory Hierarchy

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 the L3 cache.
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers.

Cache Memory Organization and Access

General Cache Concept

Cache:

Data is copied in block-sized transfer units.

Memory:

Larger, slower, cheaper memory viewed as partitioned into "blocks".

General Cache Organization (S, E, B)

Cache size:

\( C = S \times E \times B \) data bytes
**Cache Read**

E = 2^s lines per set

S = 2^b sets

B = 2^e 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

**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**

\[ M = 16 \text{ bytes (4-bit addresses), } B = 2 \text{ bytes/block,} \]
\[ S = 4 \text{ sets, } E = 1 \text{ Blocks/set} \]

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>V</th>
<th>Ta</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000]_i</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001]_i</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111]_i</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000]_i</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000]_i</td>
<td>miss</td>
</tr>
</tbody>
</table>

**Example: Direct Mapped Cache (E = 1)**

Direct mapped: One line 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
E-way Set Associative Cache (Here: E = 2)

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

Example: 2-Way Set Associative Cache Simulation

What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk

- 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)

- 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)

- Write-through + No-write-allocate
- Write-back + Write-allocate

Intel Core i7 Cache Hierarchy

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) 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.

Matrix Multiplication Example

Rearranging Loops to Improve Spatial Locality

Variable sum held 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;
  }
}
```

- 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
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 328 (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

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;
    }
}
```

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)

```c
/* 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;
    }
}
```

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 (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];
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Matrix Multiplication (ikj)

```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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

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 \(i = 0; i < n; i++\)
    - \(sum += a[i][j]\)
    - accesses successive elements
    - if block size (\(N\)) > sizeof(a[0]) bytes, explicit spatial locality
    - miss rate = \(\frac{\text{sizeof}(a)}{B}\)
- Stepping through rows in one column:
  - for \(i = 0; i < n; i++\)
    - \(sum += a[i][j]\)
    - accesses distant elements
    - no spatial locality
    - miss rate = \(\frac{1}{N \times 100\%}\)
Matrix Multiplication (jki)

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

Inner loop:
- \( A \) Column-wise
- \( B \) Fixed
- \( C \) Column-wise

Misses per inner loop iteration:
- A: 1.0
- B: 0.0
- C: 1.0

Summary of Matrix Multiplication

```c
/* kji */
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;
    }
}
```

- \( ijk & kji \):
  - 2 loads, 0 stores
  - misses/iter = 1.25

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

- \( jki & kj i\):
  - 2 loads, 1 store
  - misses/iter = 2.0

Core i7 Matrix Multiply Performance

![Graph showing performance of different matrix multiplication methods on Core i7](image)

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.