Lecture #30 – Cache Memories
Recap

• Storage technologies and trends
• Locality of reference
• Caching in the memory hierarchy
Plan for 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

Disclaimer: Slides for this lecture were borrowed from
—Randal E. Bryant and David R. O’Hallaroni’s CMU 15-213 class
Lecture Plan

• 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

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

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.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers.

Remote secondary storage (e.g., Web servers)
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:
# Examples of Caching in the Mem. Hierarchy

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>What is Cached?</th>
<th>Where is it Cached?</th>
<th>Latency (cycles)</th>
<th>Managed By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4-8 bytes words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-Chip TLB</td>
<td>0</td>
<td>Hardware MMU</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L1</td>
<td>4</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On-Chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4-KB pages</td>
<td>Main memory</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>Parts of files</td>
<td>Main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Disk cache</td>
<td>Disk sectors</td>
<td>Disk controller</td>
<td>100,000</td>
<td>Disk firmware</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>
General Cache Concepts

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

Data is copied in block-sized transfer units.

Larger, slower, cheaper memory viewed as partitioned into “blocks.”
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache: **Hit!**
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache
• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)
Types of Cache Misses

• **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.

• **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

• **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
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)

**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:

\[
\begin{array}{c}
\text{t bits} \\
\text{s bits} \\
\text{b bits}
\end{array}
\]

- tag
- set index
- block offset

data begins at this offset

B = 2^b bytes per cache block (the data)

v valid bit
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:

```
| t bits | 0...01 | 100 |
```

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

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

Address of int:

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
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>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0002</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>0012</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>0112</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>1002</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>0002</td>
<td>miss</td>
</tr>
</tbody>
</table>

Set 0

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
</tbody>
</table>

Set 1

Set 2

Set 3

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[6-7]</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:

```
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:

match: yes = hit

block offset

compare both

valid? +

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
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), ...
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>Set 0</th>
<th>Tag</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

t = 2, s = 1, b = 1

- 0: [00002], miss
- 1: [00012], hit
- 7: [01112], miss
- 8: [10002], miss
- 0: [00002], hit
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
Lecture Plan

• 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.
Memory Mountain Test Function

```c
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of array "data" with stride of "stride", 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
Lecture Plan

• Cache organization and operation

• Performance impact of caches
  • The memory mountain
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal 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;
    }
}
```

matmult/mm.c
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:
  \[
  \text{for (i = 0; i < N; i++)}\ \\
  \text{sum += a[0][i];}\ \\
  \text{– accesses successive elements}\ \\
  \text{– if block size (B) > sizeof(aij) bytes, exploit spatial locality}\ \\
  \text{miss rate = sizeof(aij) / B}
  \]

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

/* 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:

A  B  C
0.25  1.0  0.0

Inner loop:
(A,*)  (i,j)  (ii)
Row-wise  Column-wise  Fixed
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></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>
/* 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 (ikj)

/* 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)

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

/* 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>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
Summary of 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;
    }
}

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

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

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

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

- Cycles per inner loop iteration vs. Array size (n)
- Graphs for different matrix multiply operations:
  - ijk / jik
  - jki / kji
  - kij / ikj

Array size (n) ranges from 50 to 700.
Lecture Plan

• Cache organization and operation

• Performance impact of caches
  • The memory mountain
  • Rearranging loops to improve spatial locality
  • Using blocking to improve temporal locality
Example: Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

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

• Assume
  • Matrix elements are doubles
  • Cache block = 8 doubles
  • Cache size C << n (much smaller than n)

• First iteration:
  • n/8 + n = 9n/8 misses

• Afterwards in cache: (schematic)
Cache Miss Analysis

• Assume
  • Matrix elements are doubles
  • Cache block = 8 doubles
  • Cache size C << n (much smaller than n)

• Second iteration:
  • Again:
    \[ \frac{n}{8} + n = \frac{9n}{8} \text{ misses} \]

• Total misses:
  • \[ 9n/8 \times n^2 = (9/8) \times n^3 \]
Blocked Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
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)
                /* B x B mini matrix multiplications */
                    for (i1 = i; i1 < i+B; i++)
                        for (j1 = j; j1 < j+B; j++)
                            for (k1 = k; k1 < k+B; k++)
                                c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}

matmult/bmm.c
Cache Miss Analysis

• Assume
  • Cache block = 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)
Cache Miss Analysis

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

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

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

- No blocking: \((9/8) \times n^3\)
- Blocking: \(1/(4B) \times n^3\)

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

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

• 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.

Next time: Debugging and Design