Cache Memories

15-213: Introduction to Computer Systems
11th Lecture, Oct. 1, 2013

Instructors:
Randy Bryant, Dave O’Hallaron, and Greg Kesden
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
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
An Example Memory Hierarchy

- **L0:** Registers
- **L1:** L1 cache (SRAM)
- **L2:** L2 cache (SRAM)
- **L3:** Main memory (DRAM)
- **L4:** Local secondary storage (local disks)
- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

**Smaller, faster, costlier per byte**

- L1 cache holds cache lines retrieved from L2 cache
- L2 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 network servers

**Larger, slower, cheaper per byte**

- CPU registers hold words retrieved from L1 cache

---

Carnegie Mellon
General Cache Concept

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”
Many types of caches

Examples
- Hardware: L1, L2, L3 cache memories, TLBs, ...
- Software: Virtual memory, FS buffers, Web browser caches, ...

Hardware cache memories
- Significant impact on program performance
- Topic of today’s lecture
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, then in main memory**
- **Typical system structure:**

![Diagram of a typical system structure with CPU chip, Cache memory, Register file, ALU, Bus interface, System bus, I/O bridge, Memory bus, Main memory]
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:
- \( t \) 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

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

- Direct mapped:
  - One line per set

Assume:
- Cache block size 8 bytes

Address of int:
- Tag bits 0...1 100

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

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

![Diagram of direct mapped cache with valid? and match: assume yes = hit, address of int, t bits, and 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

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

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>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</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>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:

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

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

Valid? + match: yes = hit

Address of short int:

compare both

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

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):

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

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

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

L2 unified cache: 256 KB, 8-way, Access: 11 cycles

L3 unified cache: 8 MB, 16-way, Access: 30-40 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 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:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)
**Lets 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
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.
Memory Mountain Test Function

```c
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                     /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```
The Memory Mountain

- Intel Core i7
- 32 KB L1 i-cache
- 32 KB L1 d-cache
- 256 KB unified L2 cache
- 8M unified L3 cache
- All caches on-chip

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)
The Memory Mountain

Read throughput (MB/s)

Stride (x8 bytes)

Working set size (bytes)

Slopes of spatial locality

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
The Memory Mountain

Working set size (bytes)

Stride (x8 bytes)

Read throughput (MB/s)

Ridges of temporal locality

Slopes of spatial locality

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache
All caches on-chip
Today

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Line size = 32B (big enough for four 64-bit words)
  - 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 Example

**Description:**
- Multiply N x N matrices
- \( O(N^3) \) 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;
    }
}
```

Variable `sum` held in register
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 \((B) > 4\) bytes, exploit spatial locality
    - miss rate = \(4\) bytes / \(B\)
- **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;  
    }
}
```

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 (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
    }
}

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)

/* 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:

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)

```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>Misses</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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

ijo (& 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

Cycles per inner loop iteration vs. Array size (n)

- jki / kji
- ijk / jik
- kij / ikj
Today

- 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);

/* 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++)
        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 $C \ll n$ (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 $C \ll n$ (much smaller than $n$)

■ **Second iteration:**
  - Again:
    $\frac{n}{8} + n = \frac{9n}{8}$ misses

■ **Total misses:**
  - $\frac{9n}{8} \times n^2 = \left(\frac{9}{8}\right) \times n^3$
Blocked Matrix Multiplication

```c
double *c = 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];
}
```

![Block size B x B](image)
Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (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
Cache Summary

- Cache memories can have significant performance impact
- You can write your programs to exploit this!