Cache Memories:
Why Programmers Need to Know!

Instructors:
Alvin R. Lebeck
Slides from Randy Bryant and Dave O’Hallaron
Administrative

- Homework #6 Due April 11
- Y86 Simulator- groups of two (email me group members)
  - Start this ASAP, get questions out of the way.
- No Recitation Friday: Lebeck move Thur office hrs to Friday 10am. (this week only)

Today

- Review: Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
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 caches** (e.g., L1, L2, and L3), then in main memory.
- **Typical system structure:**

![Diagram of computer system components]

- CPU chip
- Register file
- ALU
- Cache memories
- Bus interface
- System bus
- Memory bus
- I/O bridge
- Main memory
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

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

E = \(2^e\) lines per set

S = \(2^s\) sets

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

- tag
- set index
- block offset

data begins at this offset

valid bit

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

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

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

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

Address of int: \( t \text{ bits} \ 0...01 \ 100 \)

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

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

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

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

Address of int:
0 1 2 3 4 5 6 7

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

No match: old line is evicted and replaced
Direct-Mapped Cache Simulation

M=16 byte 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>Set</th>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>M[6-7]</td>
</tr>
</tbody>
</table>
A Higher Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}

int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];

    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles

blackboard
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:

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

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

Address of short int:

compare both

valid? + match: yes = hit

block offset

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>
</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>10</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

The trace entries and their corresponding hits and misses are:

- 0: Miss
- 1: Hit
- 7: Miss
- 8: Miss
- 0: Hit
A Higher Level Example

```c
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache, a[0][0] goes here

32 B = 4 doubles

blackboard
What about writes?

- **Multiple copies of data exist:**
  - L1, L2, 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 immediately to memory)

- **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.

L3 unified cache (shared by all cores)
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = $1 - \text{hit rate}$
  - 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!)
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.
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

/* 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
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

Slopes of spatial locality
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

Ridges of Temporal locality
Slopes of spatial locality
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;
    }
}
```
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
    - compulsory 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!
    - compulsory 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;
    }
}
```

Inner loop:

- **Row-wise**
- **Column-wise**
- **Fixed**

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)

```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>
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></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses</td>
<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: |
|---|---|---|
| A | B | C |
| 0.0 | 0.25 | 0.25 |

Inner loop:

- Fixed
- Row-wise
- Row-wise
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></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
```

Inner loop:
- Column-wise
- Fixed
- Column-wise
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;
    }
}

Inner loop:

Column-wise
Fixed
Column-wise

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

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 (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];
    }
}

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;
    }
}

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

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 $C << n$ (much smaller than $n$)

Second iteration:
- Again: $n/8 + n = 9n/8$ misses

Total misses:
- $9n/8 \times n^2 = (9/8) \times n^3$
Blocked Matrix Multiplication

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

![Diagram of matrix multiplication with blocking]
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
- $2n/B \cdot B^2/8 = nB/4$

Total misses:
- $nB/4 \cdot (n/B)^2 = n^3/(4B)$
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
Concluding Observations

- **Programmer can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- **All systems favor “cache friendly code”**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)