Cache Memories:
Why Programmers Need to Know!

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

Administrative
- Compsci 104-16 processor project
  - Start this ASAP, get questions out of the way.

Today
- Review
- Performance impact of caches
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
  - Data layout changes to improve 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:**

![System Architecture Diagram]

---

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

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

S = 2^s sets

E = 2^e lines per set

Address of word: 

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>set</td>
<td>block</td>
</tr>
<tr>
<td>index</td>
<td></td>
<td>offset</td>
</tr>
</tbody>
</table>

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

Address of int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>s bits</th>
<th>b bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...01</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Int (4 Bytes) is here

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

v  Tag  Block

Set 0 1 0  M[0-1]  
Set 1  
Set 2  
Set 3 1 0  M[6-7]  

Mapping Arrays to Memory

Row-major  Column major

0 1 2 3 4 0 5 10 15 20
5 6 7 8 9 1 6 11 16 21
10 11 12 13 14 2 7 12 17 22
15 16 17 18 19 3 8 13 18 23
20 21 22 23 24 4 9 14 19 24

Part of the Row maps into cache
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 \( i = 0; i < N; i++ \)
    \[ \text{sum} += a[0][i]; \]
  - 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 \( i = 0; i < n; i++ \)
    \[ \text{sum} += a[i][0]; \]
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

A Higher Level DM 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;
}
```

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

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

- **find set**
  - **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>
```

- **compare both**
  - **valid?**
  - **match**: yes = hit

- **block offset**

```
<table>
<thead>
<tr>
<th>tag</th>
<th>01234567</th>
</tr>
</thead>
</table>
```

- **Additional notes**:
  - **Address of short int**:
    - **t bits**: 0…01
    - **100**

```
<table>
<thead>
<tr>
<th>tag</th>
<th>01234567</th>
</tr>
</thead>
</table>
```

- **Diagram**:
  - **E-way Set Associative Cache**
    - **Here: E = 2**
    - **E = 2**: Two lines per set
    - **Assume**: cache block size 8 bytes

```
<table>
<thead>
<tr>
<th>tag</th>
<th>01234567</th>
</tr>
</thead>
</table>
```

- **Diagram notes**:
  - **find set**
  - **E = 2**: Two lines per set
  - **Assume**: cache block size 8 bytes

```
<table>
<thead>
<tr>
<th>tag</th>
<th>01234567</th>
</tr>
</thead>
</table>
```
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):

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

<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>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

Set 0

Set 1

<table>
<thead>
<tr>
<th>v</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>
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;
}
```

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

32 B = 4 doubles

Ignore the variables \( \text{sum}, i, j \)

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

... (shared by all cores)

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.
**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: was increasing!)

---

**Cache Performance**

\[
\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time}
\]

\[
\text{Memory stall clock cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}
\]

**Example**

- Assume every instruction takes 1 cycle
- Miss penalty = 20 cycles
- Miss rate = 10%
- 1000 total instructions, 300 memory accesses
- Memory stall cycles? CPU clocks?
Cache Performance

- Memory Stall cycles = 300 \times 0.10 \times 20 = 600
- CPUclocks = 1000 + 600 = 1600

- 60% slower because of cache misses!

- Change miss penalty to 100 cycles
- CPUclocks = 1000 + 3000 = 4000 cycles

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.
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;
    }
}
```
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

Array size (n) vs. Cycles per inner loop iteration graph
Improving Data Cache Performance

- **Instruction Sequencing**
  - **Loop Interchange**: change nesting of loops to access data in order stored in memory (ijk vs kij, etc.)
  - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
  - **Blocking**: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down entire columns or rows

- **Data Layout**
  - **Merging Arrays**: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - **Nonlinear Array Layout**: Mapping 2 dimensional arrays to the linear address space
  - **Pointer-based Data Structures**: node-allocation

---

Loop Fusion Example

```c
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        { a[i][j] = 1/b[i][j] * c[i][j];
          d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access
Example: Matrix Multiplication

```c
    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];
    }
```

Blocked (Tiled) Matrix Multiplication

```c
    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];
    }
```
Blocked MXM Summary

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

- Suggest largest possible block size \(B\), but limit \(3B^2 < \text{Cache size}\!\!\)

- **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

---

**MMM Plot: Analysis**

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz

- **Reason for 20x:** Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
- **Effect:** fewer register spills, L1/L2 cache misses, and TLB misses
Data Layout Optimizations

- So far program control
- Changes the order in which memory is accessed

- We can also change the way our data structures map to memory
- 2-dimensional array
- Pointer-based data structures

Merging Arrays Example

```c
/* Before */
int val[SIZE];
int key[SIZE];

/* After */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key
Layout and Cache Behavior

- Tile elements spread out in memory because of column-major mapping
- Fixed mapping into cache
- Self-interference in cache

Making Tiles Contiguous

- Elements of a quadrant are contiguous
- Recursive layout
- Elements of a tile are contiguous
- No self-interference in cache
Pointer-based Data Structures

- Linked List, Binary Tree
- Basic idea is to group linked elements close together in memory
- Need relatively static traversal pattern
- Or could do it during garbage collection/compaction

Reducing I-Cache Misses by Compiler Optimizations

- Instructions
  - Reorder procedures in memory to reduce misses
  - Profiling to look at conflicts
Concluding Observations

- **Programmer can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking (tiling) 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)

Cache Memory Summary

- **Cost Effective Memory Hierarchy**
- **Work by exploiting locality (temporal & spatial)**
- **Associativity, Blocksize, Capacity (ABCs of caches)**
- **Know how a cache works**
  - Break address into tag, index, block offset
- **Know how to draw a block diagram of a cache**
- **CPU cycles/time, Memory Stall Cycles**
- **Your programs and cache performance**

**Next Time**

- **Exceptions and Interrupts**
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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<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:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<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];
    }
}

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></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>
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)

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

- **Total misses:**
  - $9n/8 \times n^2 = (9/8) \times n^3$
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:**
  - $\frac{B^2}{8}$ misses for each block
  - $2n/B \times \frac{B^2}{8} = nB/4$
    (omitting matrix $c$)

  - Afterwards in cache
    (schematic)

- **Second (block) iteration:**
  - Same as first iteration
  - $2n/B \times \frac{B^2}{8} = nB/4$

- **Total misses:**
  - $\frac{nB}{4} \times \frac{(n/B)^2}{4B} = n^3/(4B)$