CS 33

Caches
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:
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

\[ E = 2^e \text{ lines per set} \]

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

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

- tag
- set index
- block offset

- data begins at this offset

\[ B = 2^b \text{ bytes per cache block (the data)} \]

- valid bit

\[ v \quad \text{tag} \quad \begin{array}{c} 0 \ 1 \ 2 \ \cdots \ B-1 \end{array} \]
Example: Direct Mapped Cache (E = 1)

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

Address of int: 0...01 100

v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7
v tag 0 1 2 3 4 5 6 7

S = 2^s sets

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

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

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

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

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>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>
## 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;
}
```

```c
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
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;
}
```

```c
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;
}
```
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; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

32 B = 4 doubles
Conflict Misses: Aligned

def dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;

    for (i=0; i<8; i++)
        sum += x[i] * y[i];

    return sum;
}
Different Alignments

```c
double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;
    for (i=0; i<8; i++)
        sum += x[i] * y[i];
    return sum;
}
```

32 B = 4 doubles
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
E-way Set-Associative Cache (Here: E = 2)

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

Address of short int:

\[ t \text{ bits} \quad 0...01 \quad 100 \]

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), ...
Given the address above and the cache contents as shown, what is the value of the *int* at the given address?

a) 1111  
b) 3333  
c) 4444  
d) 7777
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 Index</th>
<th>Address</th>
<th>Status</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>hit</td>
</tr>
</tbody>
</table>

<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>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td></td>
<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;
}
```

Ignore the variables sum, i, j

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

32 B = 4 doubles
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;
}
```

```c
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

32 B = 4 doubles
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

32 B = 4 doubles
Conflic Misses

double dotprod(double x[8], double y[8]) {
    double sum = 0.0;
    int i;
    for (i=0; i<8; i++)
        sum += x[i] * y[i];
    return sum;
}
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache
- L3 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
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
Cache Performance Metrics

• **Miss rate**
  - fraction of memory references not found in cache
    (misses / accesses)
    \[\text{miss rate} = 1 - \text{hit rate}\]
  - 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 cycles 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% hit rate is twice as good as 97%?
  – consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  – average access time:
    97% hits: \(0.97 \times 1 \text{ cycle} + 0.03 \times 100 \text{ cycles} \approx 4 \text{ cycles}\)
    99% hits: \(0.99 \times 1 \text{ cycle} + 0.01 \times 100 \text{ cycles} \approx 2 \text{ cycles}\)

• 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
Miss-Rate Analysis for Matrix Multiply

• Assume:
  – Block size = 32B (big enough for four 64-bit words)
  – matrix dimension (N) is very large
    » approximate 1/N as 0.0
  – cache is not big enough to hold multiple rows

• Analysis method:
  – look at access pattern of inner loop

\[
C_{ij} = A_{ik} \times B_{kj}
\]
Matrix Multiplication Example

• Description:
  – multiply N x N matrices
  – O(N³) 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:
  – \( \textbf{for} \ (i = 0; \ i < N; \ i++) \)
    \begin{align*}
    \text{sum} &= \text{sum} + \text{a}[0][i]; \\
    \end{align*}
  – accesses successive elements
  – if block size (B) > 4 bytes, exploit spatial locality
    » compulsory miss rate = 4 bytes / B
• Stepping through rows in one column:
  – \( \textbf{for} \ (i = 0; \ i < n; \ i++) \)
    \begin{align*}
    \text{sum} &= \text{sum} + \text{a}[i][0]; \\
    \end{align*}
  – 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;
    }
}
```

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>

Inner loop: 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>misses</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];
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Misses per inner loop iteration:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
</tr>
</tbody>
</table>

Inner loop:

- **(i,k)**: Row-wise
- **(k,*)**: Row-wise
- **(i,*)**: Fixed

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Row-wise</td>
<td>Fixed</td>
<td>Row-wise</td>
</tr>
<tr>
<td>k</td>
<td>Row-wise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CS33 Intro to Computer Systems  
XVIII–32
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];
    }
}
```

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>0.0</td>
<td>0.25</td>
<td>0.25</td>
<td></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>1</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

\[
\begin{align*}
\text{ijk} & : \\
& \quad \cdot \ 2 \ \text{loads}, \ 0 \ \text{stores} \\
& \quad \cdot \ \text{misses/iter} = 1.25 \\
\text{jki} & : \\
& \quad \cdot \ 2 \ \text{loads}, \ 1 \ \text{store} \\
& \quad \cdot \ \text{misses/iter} = 2.0 \\
\end{align*}
\]

\[
\begin{align*}
\text{kij} & : \\
& \quad \cdot \ 2 \ \text{loads}, \ 1 \ \text{store} \\
& \quad \cdot \ \text{misses/iter} = 0.5 \\
\end{align*}
\]
Core i7 Matrix Multiply Performance

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

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

CS33 Intro to Computer Systems
XVIII–37
Matrix Multiplication: More Analysis

/* 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 << 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:
  – \[ 9\frac{n}{8} \times n^2 = \left(\frac{9}{8}\right) \times n^3 \]
Blocked Matrix Multiplication

```c
/* 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
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
  – $2n/B \cdot B^2/8 = nB/4$ (omitting matrix $c$)
  – afterwards in cache (schematic)

$$
\begin{array}{c}
\text{Block size } B \times B \\
\text{n/B blocks}
\end{array}
$$
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:
  – $nB/4 \times (n/B)^2 = n^3/(4B)$
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 element used \( O(n) \) times!
  – but program has to be written properly
Quiz 2

What is the smallest value of B (in 8-byte doubles) for which the cache-miss analysis works?

a) 1  
b) 2  
c) 4  
d) 8
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)