Cache Memories

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron
Topics

- Cache memory organization and operation
- Performance impact of caches
**Cache Memories**

- **Cache memories** are small, fast SRAM-based memories
  - Hold frequently accessed blocks of main memory
  - Managed automatically in hardware

- **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.**

- **Typical system structure:**

![Diagram of a typical system structure with CPU chip, Register file, ALU, System bus, Memory bus, I/O bridge, and Main memory.]
General Cache Organization \((S, E, B)\)

- **S** = number of sets \((2^s)\)
- **E** = lines per set \((2^e)\)
- **B** = bytes per cache block (the data) \((2^b)\)

**Cache size:**
\[ C = S \times E \times B \text{ data bytes} \]
Cache Read

- Locate set
- Check if any line in set has matching tag
- If Match + 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

MSB memory address \((\text{address size} - s - b)\)

- valid bit

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

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

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

Address of int:

| t bits | 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

On cache miss: old line is evicted and replaced
Direct-Mapped (E=1) Cache Simulation

M=16 byte 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>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>

<table>
<thead>
<tr>
<th>Set</th>
<th>v</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 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 (stored in registers)

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

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

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

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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 (stored in registers)
assume: cold (empty) cache, a[0][0] goes here

Miss ratio = ?

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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 (stored in registers)
assume: cold (empty) cache, a[0][0] goes here

Miss ratio = .25
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 (stored in registers)
assume: cold (empty) cache, a[0][0] goes here

![Cache diagram with variables and example code]

Miss ratio = ?

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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;
}
```

Miss ratio = ?

Ignore the variables sum, i, j (stored in registers)

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

Miss ratio = ?

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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;
}
```

Miss ratio = 0.25

Ignore the variables sum, i, j (stored in registers)

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

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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` (stored in registers)

a[0][0] swapped for a[2][0]

Miss ratio = 0.25

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

Ignore the variables sum, i, j (stored in registers)

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

Miss ratio = ?

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

Miss ratio = 1.0

```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 (stored in registers)
assume: cold (empty) cache, a[0][0] goes here

Miss ratio = 1.0

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
**A Higher Level Example**

Miss ratio = ?

```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` (stored in registers)

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

![Array Visualization](image)

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

Miss ratio = 1.0

```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 \(\text{sum}, i, j\) (stored in registers)

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

Miss ratio = 1.0

![Matrix diagram]

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

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

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

Miss ratio = ?

```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;
}
```

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

Miss ratio = 1.0

```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 (stored in registers)
assume: cold (empty) cache, a[0][0] goes here

Miss ratio = 1.0

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
A Higher Level Example

Ignore the variables sum, i, j (stored in registers)
a[0][0] is swapped with a[8][0]

Miss ratio = 1.0

```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;
}
```

32 B = 4 doubles (line size)

Administrative flag and tag bits not shown
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 | 01 | 100 |
```

find set

```
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
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
```
E-way Set Associative Cache (Here: \( E = 2 \))

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

Address of short int:

```
01 100
```

t bits

compare both

valid? + match: yes = hit

block offset

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

```
v tag 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

Address of short int:

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

valid? + match: yes = hit

compare both

block offset

short int (2 Bytes) is here

On cache miss:
• One line in set is selected for eviction and replacement
• Replacement policies: random, least recently used (LRU), ...
# 2-Way Set Associative Cache Simulation

- $t=2$\;\;s=1\;\;b=1$

<table>
<thead>
<tr>
<th>xx</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
</table>

- 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>[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>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></td>
<td>10</td>
<td>M[8-9]</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>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
E-Way Set Associative Cache Simulation

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB
- D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
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** (write to memory on eviction)
    - Dirty bit (keeps track if line is different from memory)

- **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 CPU 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

i – instruction
d – data
Topics

- Cache organization and operation
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
Cache Performance

- **Miss Rate**
  - Fraction of memory references not found in cache
  - Typical numbers (in percentages):
    - 3-10% miss rate for L1
    - can be quite small (e.g., < 1%) for L2, depending on L2 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 (~10x L1 hit time)

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)
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**
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: Locality is quantified through understanding of cache memories.
The Memory Mountain

Based on Intel Core i7

Read throughput (MB/s)

Element size
(x8 byte strides)

Data size (bytes)
The Memory Mountain

Based on Intel Core i7

Slopes of spatial locality
The Memory Mountain

Based on Intel Core i7

Ridges of Temporal locality

Slopes of spatial locality

Read throughput (MB/s)

Element size (x8 byte strides)

Data size (bytes)
Topics

- 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
    - Cache size > 1 row
    - Cache size < 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

```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;
    }
}
```

Variables i, j, sum held in registers
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:**
  - accesses successive elements
  - exploit spatial locality!

- **Stepping through rows in one column:**
  - accesses distant elements
  - no spatial locality!
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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Inner loop:

- **A**: Fixed
- **B**: Row-wise
- **C**: Row-wise

Diagram:

- **(i,k)**
- **(k,*)**
- **(i,*)**
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;
    }
}
```

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
Topics

- Cache organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
Using blocking to improve temporal locality

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

/* Example: 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:
Cache Miss Analysis

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

Second iteration:
- Again in cache: \\
  \( n/8 + n = 9n/8 \) misses

Total misses:
- \( 9n/8 * n^2 = (9/8) * n^3 \)
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];
}

Block size B x B
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
**Cache Miss Analysis**

- **Assume:**
  - Cache block = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $\Box$ 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 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)