Cache Performance

CMPU 224 – Computer Organization
Jason Waterman
The following table gives the parameters for a number of different caches, where \( m \) is the number of address bits, \( C \) is the cache size (number of data bytes), \( B \) is the block size in bytes, and \( E \) is the number of lines per set. For each cache, determine the number of cache sets \( (S) \), tag bits \( (t) \), set index bits \( (s) \), and block offset bits \( (b) \).

<table>
<thead>
<tr>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


The following table gives the parameters for a number of different caches, where \( m \) is the number of address bits, \( C \) is the cache size (number of data bytes), \( B \) is the block size in bytes, and \( E \) is the number of lines per set. For each cache, determine the number of cache sets (\( S \)), tag bits (\( t \)), set index bits (\( s \)), and block offset bits (\( b \)).

<table>
<thead>
<tr>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
C = S \times E \times B \\
S = \frac{C}{E \times B} \\
m = t + s + b \\
t = m - (s + b)
\]
The following table gives the parameters for a number of different caches, where $m$ is the number of address bits, $C$ is the cache size (number of data bytes), $B$ is the block size in bytes, and $E$ is the number of lines per set. For each cache, determine the number of cache sets ($S$), tag bits ($t$), set index bits ($s$), and block offset bits ($b$).

<table>
<thead>
<tr>
<th>$m$</th>
<th>$C$</th>
<th>$B$</th>
<th>$E$</th>
<th>$S$</th>
<th>$t$</th>
<th>$s$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>4</td>
<td>64</td>
<td>24</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$C = S \times E \times B$$
$$S = \frac{C}{E \times B}$$
$$m = t + s + b$$
$$t = m - (s + b)$$
Cache Practice

The following table gives the parameters for a number of different caches, where \( m \) is the number of address bits, \( C \) is the cache size (number of data bytes), \( B \) is the block size in bytes, and \( E \) is the number of lines per set. For each cache, determine the number of cache sets (\( S \)), tag bits (\( t \)), set index bits (\( s \)), and block offset bits (\( b \)).

<table>
<thead>
<tr>
<th>( m )</th>
<th>( C )</th>
<th>( B )</th>
<th>( E )</th>
<th>( S )</th>
<th>( t )</th>
<th>( s )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>4</td>
<td>64</td>
<td>24</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>4</td>
<td>256</td>
<td>1</td>
<td>30</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>1</td>
<td>128</td>
<td>22</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>8</td>
<td>128</td>
<td>1</td>
<td>29</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>4</td>
<td>8</td>
<td>24</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
C = S \times E \times B \\
S = \frac{C}{E \times B} \\
m = t + s + b \\
t = m - (s + b)
\]
Today

• Cache performance metrics

• The graph on the cover of your textbook explained

• Writing cache friendly code
What about writes?

• Multiple copies of the data exist:
  • Cache and Main Memory

• What to do on a write-hit?
  • Update cache block with new contents
  • Write-through (write immediately to memory)
  • Write-back (defer write to memory until line is evicted)
    • Need a modified bit (whether line is different from memory or not)

• What to do on a write-miss?
  • No-write-allocate (writes straight to memory, does not load into cache)
  • Write-allocate (load into cache, update line in cache)
    • Good if more writes to the location follow

• Typical Pairings
  • Write-through + No-write-allocate
    • Simpler
  • Write-back + Write-allocate
    • Better performance
Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty

- **Conflict miss**
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same set in the cache
    - E.g., referencing blocks 0, 8, 0, 8, 0, 8 in our direct-mapped example would miss every time
    - If the cache were fully associative, this access pattern wouldn’t be a miss

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache
Cache Performance Metrics

• Miss Rate
  • Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  • Typical numbers:
    • 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:
    • 4 clock cycles for L1
    • 10 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 typically used instead of “hit rate”
  • 3% versus 1%
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
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
long data[MAXELEMS]; /* Global array to traverse */

/* test - Iterate over first "elems" elements of array “data”
* with stride of "stride", using using 4x4 loop unrolling. */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i += stride) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Call test() with many combinations of elems and stride.

For each elems and stride:

1. Call test() once to warm up the caches
2. Call test() again and measure the read throughput (MB/s)
The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size
Matrix Multiplication Example

- **Description:**
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
    - 2*N reads per source element
    - $N^2$ elements
  - 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
Miss Rate Analysis for Matrix Multiply

• Assume:
  • Block size = 32B (big enough for four doubles)
  • Matrix dimension (N) is very large
  • Cache is not even big enough to hold multiple rows

• Analysis Method:
  • Look at access pattern of inner loop

\[ C_{i,j} = A_{i,k} \times B_{k,j} \]
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) > sizeof(a_{ij}) bytes, exploit spatial locality
    • miss rate = sizeof(a_{ij}) / 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>Misses</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;
    }
}
```

**Inner loop:**
- **Row-wise Misses:** 0.25
- **Column-wise Misses:** 1.0
- **Fixed Misses:** 0.0

**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];
    }
}
```

**Inner loop:**
- \((i,k)\)
- \((k,*)\)
- \((i,*\)

**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)

/* 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:

Fixed    Row-wise  Row-wise
A         B          C

Misses per inner loop iteration:

<table>
<thead>
<tr>
<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>
</tr>
</tbody>
</table>
Matrix Multiplication (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;
    }
}
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

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++)
        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
Cache Summary

• Cache memories can have significant performance impact

• You can write your programs to exploit this!
  • Focus on the inner loops, where bulk of computations and memory accesses occur
  • Try to maximize spatial locality by reading data objects sequentially with stride 1
  • Try to maximize temporal locality by using a data object as often as possible once it’s read from memory