Caches Part 2

CMPU 224 – Computer Organization
Jason Waterman
Today

• Cache performance metrics

• The graph on the cover of your textbook explained

• Writing cache friendly code
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:
$C = S \times E \times B$ data bytes
Cache Read

1. Locate set
2. Check if any line in set has matching tag
3. Yes + line valid: hit
4. Locate data starting at offset

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

tag
set index
block offset
data begins at this offset

E = 2^e lines per set

S = 2^s sets

B = 2^b bytes per cache block (the data)

valid bit
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 dirty 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, it 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 cycle 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 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)
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

```
\[
\begin{bmatrix}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
\end{bmatrix}
\times
\begin{bmatrix}
  7 & 8 \\
  9 & 10 \\
  11 & 12 \\
\end{bmatrix}
= \begin{bmatrix}
  58 \\
\end{bmatrix}
\]
```

$1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58$
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>A</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>

**Inner loop:**

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

A

B

C

(i,*), (*,j), (i,j)
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>

Inner loop:

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

(i,*): Row-wise
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:**
- **(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></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;
    }
}
```

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>
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;
    }
}

Misses per inner loop iteration:

\begin{array}{ccc}
  A & B & C \\
  1.0 & 0.0 & 1.0 \\
\end{array}

Inner loop:

- Column-wise: (*)
- Fixed: (k,j)
- Column-wise: (*,j)
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;
    }
}

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

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];
    }
}

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

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;
    }
}

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0
Core i7 Matrix Multiply Performance

![Graph showing Core i7 Matrix Multiply Performance](image)

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

Array size (n)

Cycles per inner loop iteration

11/26/2019

CMPU 224 -- Computer Organization
Matrix Multiplication Revisited

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 is \( n \) by \( n \) (where \( n \) is large)
  • 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 is $n \times n$ (where $n$ is large)
  • Matrix elements are doubles
  • Cache block = 8 doubles
  • Cache size $C << n$ (much smaller than $n$)

• Second iteration:
  • Again:
    $\frac{n}{8} + n = \frac{9n}{8}$ misses

• Total misses:
  • $9n/8 \times n^2 = (9/8) \times n^3$
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];
}

matmult/bmm.c

11/26/2019
CMPU 224 -- Computer Organization Fall 2016
Cache Miss Analysis

• Assume:
  • Cache block $B = 8$ 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 $c$)

• Total misses:
  • $nB/4 \cdot (n/B)^2 = n^3/(4B)$
Blocking Summary

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

• Reason for dramatic difference:
  • Matrix multiplication has inherent temporal locality
  • But program has to be written properly
  • Blocked version is much harder to understand
    • Save for libraries or leave optimization to the compiler

• Performance improvements on a core i7 are not that great though!
  • Aggressive cache prefetching on non-blocking version
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 with sequentially with stride 1
  • Try to maximize temporal locality by using a data object as often as possible once it’s read from memory