Caches Part 2

CMPU 224 – Computer Organization
Jason Waterman
Today

• Cache performance metrics

• Writing cache friendly code

• The graph on the cover of your textbook explained
General Cache Organization (S, E, B)

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

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

E = $2^e$ lines per set

$S = 2^s$ sets

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

Address of word:

- **t bits**
- **s bits**
- **b bits**

- **tag**
- **set index**
- **block offset**

valid bit

data begins at this offset
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) {
    int 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

![Diagram](image-url)
Matrix Multiplication Example

- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
    - $2N$ reads per source element
    - $N^2$ elements
  - N values summed per destination
    - But may be able to hold in register

```
/* 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

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

\[
C_{ij} = \sum_{k} A_{ik} \times B_{kj}
\]
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) > `sizeof(a_{ij})` bytes, exploit spatial locality
    - miss rate = \( \text{sizeof}(a_{ij}) / B \)

- Stepping through rows in one column:
  - for \( i = 0; i < n; i++ \)
    \[
    \text{sum} += a[i][0];
    \]
  - 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></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>A</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></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:

- **Fixed**
- **Row-wise**
- **Row-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>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
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)**: Fixed
- **(k,*)**: Row-wise
- **(i,*)**: Row-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>Misses</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:
- (*,k) Column-wise
- (k,j) Fixed
- (*,j) 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)

```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>

Inner loop:

- **(*,k)**: Column-wise
- **(k,j)**: Fixed
- **(*,j)**: Column-wise
Summary of Matrix Multiplication

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

```c
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;
    }
}
```

```c
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];
    }
}
```

```c
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;
}
```
Core i7 Matrix Multiply Performance

![Graph showing performance of different matrix multiplication algorithms]

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

Array size (n)

Cycles per inner loop iteration

5/2/2019 CMPU 224 -- Computer Organization
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