Caching and Performance

Kai Shen

Caches (broader interpretation)

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

- Why do memory hierarchies work?
  - Because of locality, most accesses are satisfied by cache.
  - So you get the storage space of lower-level storage but enjoy the speed of higher level cache.

What Data to Keep in Caches?

- To achieve the goal that most accesses are satisfied by cache, what data to keep in cache?

  - For temporal locality
    - Keep data that is recently accessed
  
  - For spatial locality
    - Keep data that is adjacent (next) to currently accessed data
General Cache Concepts: Hit

Data in block b is needed

Block b is in cache:
Hit!

Cache

Request: 14

5

General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from lower-level

Cache

Request: 12

Replacement policy:
determines which block gets evicted (victim)

6

Caches (narrower interpretation)

- 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:

```
   +-----------------------+
   |                      |
   |  CPU chip             |
   |                      |
   |  Register file        |
   |                      |
   |  Cache memories       |
   |                      |
   |  ALU                  |
   |                      |
   |  System bus          |
   |                      |
   |  Memory bus          |
   |                      |
   |  Bus interface        |
   |                      |
   +-----------------------+
```

- E-way associative (flexibility of replacement)

7

Cache Set Association

E = 2^e lines per set

S = 2^s sets

A cache line is a basic unit of access to memory (e.g., 64 bytes)
- Each set is mapped to a certain section of the memory space (according to particular s bits in a segment of the memory address)
- E-way associative (flexibility of replacement)

8
Cache

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

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

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

Valid bit

\[ V \]

Address of word:

\[ \text{tag} \quad \text{set} \quad \text{line} \quad \text{offset} \]

E = 2^e lines per set

S = 2^s sets

Cache Helps Programs with Locality

- Cache improves the performance of programs with temporal locality
  - Replacement policy prefers to retain more recently accessed data
    (limited in direct-mapped cache)

- Cache improves the performance of programs with spatial locality
  - Cacheline-grained memory access (implicit prefetching)
  - Explicit prefetching

What about writes?

- Write-through
  - write immediately to memory

- Write-back
  - write to cache only (most of time)
  - defer write to memory until replacement of line
### Intel Core i7 Cache Hierarchy

- **Core 0**
  - Regs
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache

- **Core 3**
  - Regs
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache

- **L3 unified cache** (shared by all cores)

- **Main memory**

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

Cache line size: 64 bytes for all caches.

### Cache Performance
- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits (fraction of memory accesses in cache) 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*

- **“Miss rate” is more illustrative than “hit rate”**

### The Memory Mountain

#### Intel Core i7
- 32 KB L1 i-cache
- 32 KB L1 d-cache
- 256 KB unified L2 cache
- 8M unified L3 cache

- **All caches on-chip**

#### Read throughput (MB/s)
- Stride (x8 bytes)
- Working set size (bytes)
- Spatial locality
- Temporal locality

#### The Memory Mountain
- **Slopes of spatial locality**
- Intel Core i7
- 32 KB L1 i-cache
- 32 KB L1 d-cache
- 256 KB unified L2 cache
- 8M unified L3 cache

- **All caches on-chip**
The Memory Mountain

- Intel Core i7
- 32 KB L1 i-cache
- 32 KB L1 d-cache
- 256 KB unified L2 cache
- 8 MB unified L3 cache
- All caches on-chip

Ridges of Temporal locality
Slopes of spatial locality

Read throughput (MB/s)
Working set size (bytes)
Stride (x8 bytes)

Matrix Multiplication Performance

```
/* 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

- Multiply n x n matrices
- O(n^3) total operations
- 3n^3 data accesses
- Cache misses dominate the performance
- Some data accesses go to registers
- Some memory accesses hit in cache

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)
  - Sequential reference patterns are good (spatial locality)
- Case studies
  - Rearrange loops to improve spatial locality
  - Use blocking to improve temporal locality

Layout of C Arrays in Memory (review)

- For C arrays, each inner vector in contiguous memory locations
- Stepping through elements in one inner vector:
  - for (i = 0; i < n; i++)
    - sum += a[i][j];
    - accesses successive elements
    - recall that we load a cacheline at a time
    - if cache line size (64) > 8 bytes, exploit spatial locality
      - miss rate = 8 / 64
- Stepping through the outer index:
  - for (i = 0; i < n; i++)
    - sum += a[i][0];
    - accesses distant elements
    - no spatial locality!
      - miss rate = 1.0 (i.e. 100%)

- Variable sum held 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;
    }
}
```
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;
    }
}
```

Assume matrix dimension (n) is so large that a single row can’t fit into cache.

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ijk</td>
<td>0.125</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Other implementations?

Matrix Multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = a[i][k];
        for (i=0; i<n; i++)
            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>kij</td>
<td>0.0</td>
<td>0.125</td>
<td>0.125</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>jki</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Summary of Matrix Multiplication

- **ijk (k & jik):**
  - misses/iter = 1.125
- **kij (k & ikj):**
  - misses/iter = 0.25
- **jki (k & jkj):**
  - misses/iter = 2.0
Core i7 Matrix Multiply Performance

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

corei7/matrixmultiply.png

Blocked Matrix Multiplication

```c
void mmr(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)
                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];
}
```

Cache Miss Analysis

- Assume:
  - Cache line = 8 doubles (elements)
  - Three blocks fit into cache
  - B is the block size in number of elements
  - n/B blocks

- First (block) iteration:
  - B^2/8 misses for each block
  - 2n/B * B^2/8 = nb/4
    (omitting matrix C)

- Total misses
  - nb /4 * [n/B]^2 = n^3/(4B)

Summary

- Blocking: \( \frac{n^3}{4B} \)
- No blocking: \( n^3 \times 0.25 \)

- Suggest largest possible block size B, but limit it so three blocks fit into cache
  - I once used block size B=25 elements (doubles), what was the cache size?

- Foundation for performance enhancement:
  - 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
Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.