Cache Memories

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:

![System Structure Diagram]

Cache Read

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

- Locate set
- Check if any line in set has matching tag
- Yes = line valid: hit
- Locate data starting at offset

Example: Direct Mapped Cache (E = 1)

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

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

Today

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

General Cache Organization (S, E, B)

Set
Line
\[ E = 2^e \text{ lines per set} \]
\[ S = 2^s \text{ sets} \]
\[ B = 2^b \text{ bytes per cache block (the data)} \]
Example: Direct Mapped Cache (E = 1)
Direct mapped: One line per set
Assume: cache block size 8 bytes

Address of int:

block offset

valid? + match: assume yes = hit

Example: Direct Mapped Cache (E = 1)
Direct mapped: One line per set
Assume: cache block size 8 bytes

No match: old line is evicted and replaced
E-way Set Associative Cache (Here: E = 2)
E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

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

---

Find set

```
left right
left right
left right
left right
```

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

---

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;
}
```

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

---

A Higher Level Example

```c
int sum_array_cols(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

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

---

2-Way Set Associative Cache Simulation

```
M=16 byte addresses, B=2 bytes/block, S=2 sets, E=2 blocks/set
```

Address trace (reads, one byte per read):

```
0 [0000], miss
1 [0001], hit
7 [0111], miss
8 [1000], miss
0 [0000], hit
```

---

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;
}
```

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

---

A Higher Level Example

```c
int sum_array_cols(double a[16][16]) {
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Assume: cold (empty) cache, a[0][0] goes here
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;
}
```

```c
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;
    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

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 (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- 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

Administrative Break

- We’ve posted several updates about the architecture lab
  - Download new materials from the Moodle
  - Check the forum for some clarifications

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
    - 1 – hit rate
  - Typical numbers (in percentages):
    - 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:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Intel Core i7 Cache Hierarchy

Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 99% hits vs. 97% hits?
  - Consider:
    - cache hit time of 1 cycle
    - miss penalty of 100 cycles
  - What’s the ratio of average access times?
  - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
  - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles
  - 99% hit rate is twice as fast!
- Moral: this is why “miss rate” is used instead of “hit rate”
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.

Today

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

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
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;
    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn’t optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz) {
    double cycles;
    int elems = size / sizeof(int);
    test(elems, stride);                     /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

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
Administrative Break
- We’ve heard requests for postponing the lab 4 due date
  - We’re thinking about it
  - Watch for a decision announced tomorrow
- Assignment 4, on caches, out tonight
- Cache lab out next week

Today
- 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
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold 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
  - N values summed per destination
    - but may be able to hold in register

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[i][j];
    - accesses successive elements
    - if block size (B) > 4 bytes, exploit spatial locality
      - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    - sum += a[i][0];
    - accesses distant elements
    - no spatial locality!
      - compulsory 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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.00</td>
<td>0.00</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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1.00</td>
<td>0.00</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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</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;
    }
}
```

Misses per inner loop iteration:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>1.00</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>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Summary of Matrix Multiplication

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

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

- **jki (i & kji):**
  - 2 loads, 1 store
  - misses/iter = 2.0

Core i7 Matrix Multiply Performance

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

Today

Example: Matrix Multiplication

```c
double *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 elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - n/8 + n = 9n/8 misses

- **Second iteration:**
  - Again:
    - n/8 + n = 9n/8 misses

- **Total misses:**
  - 9n/8 * n² = (9/8) * n³
**Blocked Matrix Multiplication**

```c
n = (double *) calloc((size_t) 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];
}
```

**Cache Miss Analysis**

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: 3B^2 < C

- **First (block) iteration:**
  - B^2/B misses for each block
  - 2n/B * 2n/B = 4nB

- **Second (block) iteration:**
  - Same as first iteration
  - 2n/B * 2n/B = 4nB

- **Total misses:**
  - 4nB

**Summary**

- **No blocking:** (9/8) * n^3
- **Blocking:** 1/(4B) * 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 element 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)