Lecture 15: Caches and Optimization

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012

Last time

- Program optimization
  - Optimization blocker: Memory aliasing
  - One solution: Scalar replacement of array accesses that are reused

```c
for (i = 0; i < n; i++) {
    b[i] = 0;
    for (j = 0; j < n; j++)
        b[i] += a[i*n + j];
}
```

```c
for (i = 0; i < n; i++) {
    double val = 0;
    for (j = 0; j < n; j++)
        val += a[i*n + j];
    b[i] = val;
}
```

Last time

- Instruction level parallelism
- Latency versus throughput

![Functional Units Diagram]

Step 1: 1 cycle
Step 2: 1 cycle
Step 10: 1 cycle

Last time

Consequence:

Twice as fast

Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations

Problem: Processor-memory bottleneck

Processor performance doubled about every 18 months
Bus bandwidth evolved much slower

Core 2 Duo:
Can process at least 256 Bytes/cycle
(1 SSE two operand add and mult)

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100 cycles

Solution: Caches
**Cache**

- **Definition:** Computer memory with short access time used for the storage of frequently or recently used instructions or data

**General cache mechanics**

- Smaller, faster, more expensive memory caches a subset of the blocks
- Larger, slower, cheaper memory viewed as partitioned into “blocks”

**General cache concepts: Hit**

- **Data in block b is needed**
- **Request:** 14
- **Cache:** 8 9 14 3
- **Memory:** 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- **Block b is in cache:** Hit!

**General cache concepts: Miss**

- **Data in block b is needed**
- **Request:** 12
- **Cache:** 8 12 14 3
- **Memory:** 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
- **Block b is not in cache:** Miss!
- **Block b is fetched from memory**
- **Placement policy:** determines where block goes
- **Replacement policy:** determines which block gets evicted (victim)

**Cache performance metrics**

- **Miss Rate**
  - Fraction of memory references not found in cache
  - 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!)

**Lets 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”
Types of cache miss

- Cold (compulsory) miss
  - Occurs on first access to a block

- Conflict miss
  - Most hardware caches limit blocks to a small subset (sometimes a singleton) of the available cache slots
    - e.g., block \( i \) must be placed in slot \( (i \mod 4) \)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

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

- Coherency miss
  - Multiprocessor systems: see later in the course

Why caches work

- Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future

- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time

Example: locality?

- Data:
  - Temporal: sum referenced in each iteration
  - Spatial: array \( a[] \) accessed in stride-1 pattern

- Instructions:
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

Being able to assess the locality of code is a crucial skill for a programmer

Locality example #1

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality example #2

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

Locality example #3

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

- How can it be fixed?
Memory hierarchies

- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True of registers $\leftrightarrow$ DRAM, DRAM $\leftrightarrow$ disk, etc.
    - Well-written programs tend to exhibit good locality
- These properties complement each other beautifully
- They suggest an approach for organizing memory and storage systems known as a memory hierarchy

Examples of caching in the hierarchy

<table>
<thead>
<tr>
<th>Cache type</th>
<th>What is cached?</th>
<th>Where is it cached?</th>
<th>Latency (cycles)</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>4/8-byte words</td>
<td>CPU core</td>
<td>0</td>
<td>Compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>Address translations</td>
<td>On-chip TLB</td>
<td>0</td>
<td>Hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>64-byte blocks</td>
<td>On-chip L1</td>
<td>1</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte blocks</td>
<td>On-chip L2</td>
<td>10</td>
<td>Hardware</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>4kB page</td>
<td>Main memory (RAM)</td>
<td>100</td>
<td>Hardware + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>4kB sectors</td>
<td>Main memory (RAM)</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>Parts of files</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>SMB/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>Web pages</td>
<td>Local disk</td>
<td>10,000,000</td>
<td>Web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>Web pages</td>
<td>Remote server disks</td>
<td>1,000,000,000</td>
<td>Web proxy server</td>
</tr>
</tbody>
</table>

Memory hierarchy: Core 2 Duo

<table>
<thead>
<tr>
<th></th>
<th>Throughput:</th>
<th>Latency:</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1/L2 cache: 64 B blocks</td>
<td>16 B/cycle</td>
<td>3 cycles</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>8 B/cycle</td>
<td>14 cycles</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>2 B/cycle</td>
<td>100 cycles</td>
</tr>
<tr>
<td>Main Memory</td>
<td>1 B/30 cycles</td>
<td>millions</td>
</tr>
</tbody>
</table>

Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations
Cache read

- 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

Example: Direct mapped cache (E = 1)

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

Example

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;
}

Caching variables: sum, i, j

E-way set-associative cache (here: E = 2)

- E = 2: Two lines per set
- Assume: cache block size 8 bytes

E-way set-associative cache (here: E = 2)

- E = 2: Two lines per set
- Assume: cache block size 8 bytes

Example

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;
}

Ignore the variables sum, i, j

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

No match: old line is evicted and replaced

E-way set-associative cache (here: E = 2)

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

Ignore the variables sum, i, j

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

```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

Software caches are more flexible

- Examples
  - File system buffer caches, web browser caches, etc.

- Some design differences
  - Almost always fully associative
    - so, no placement restrictions
  - index structures like hash tables are common
  - Often use complex replacement policies
    - misses are very expensive when disk or network involved
    - worth thousands of cycles to avoid them
  - Not necessarily constrained to single “block” transfers
    - may fetch or write-back in larger units, opportunistically

Today

- Memory hierarchy, caches, locality
- Cache organization
- Program optimization:
  - Cache optimizations

Optimizations for the memory hierarchy

- Write code that has locality
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time
- How to achieve this?
  - Proper choice of algorithm
    - Loop transformations

- Cache versus register level optimization:
  - In both cases locality desirable
  - Register space much smaller + requires scalar replacement to exploit temporal locality
  - Register level optimizations include exhibiting instruction level parallelism (conflicts with locality)

Example: matrix multiplication

```c
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 $C \ll n$ (much smaller than $n$)

- First iteration:
  - $n/8 + n = 9n/8$ misses
  - Afterwards in cache: (schematic)

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

- Total misses:
  - $9n/8 \times n^2 = (9/8) \times n^3$

Blocked matrix multiplication

```c
void mmn(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 block = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks fit into cache: $3B^2 < C$

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache: (schematic)

- Total misses:
  - $nB/4 \times (n/B)^2 = n^3/(4B)$

Summary

- No blocking: $(9/8) \times n^3$
- Blocking: $1/(4B) \times n^3$
- Suggest largest possible block size $B$, but limit $3B^2 < C$!
  (can possibly be relaxed a bit, but there is a limit for $B$)
- Reason for dramatic difference:
  - 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