Computer Systems
CSE 410 Autumn 2013
10 – Memory Organization and Caches
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
pushq   %rbp
movq    %rsp, %rbp
...
popq    %rbp
ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000001111

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac
Caches
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
Making memory accesses fast!

- What we want: Memories that are
  - Big
    - Fast
    - Cheap

- Hardware: Pick any two

- So we’ll be clever...
How does execution time grow with SIZE?

```c
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```

Plot
Actual Data
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

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

Problem: lots of waiting on memory
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

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

Solution: caches
Cache

- **English definition:** a hidden storage space for provisions, weapons, and/or treasures

- **CSE definition:** computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

more generally,

used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)
**General Cache Mechanics**

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

Data is copied in block-sized transfer units.
General Cache Concepts: **Hit**

Data in block b is needed

Block b is in cache: **Hit!**
General Cache Concepts: **Miss**

**Data in block b is needed**

**Block b is not in cache:**

**Miss!**

**Block b is fetched from memory**

**Block b is stored in cache**

- **Placement policy:** determines where b goes
- **Replacement policy:** determines which block gets evicted (victim)
Not to forget…

- CPU
- A little of super fast memory (cache$)
- Lots of slower Mem
Memory and Caches

- Cache basics
- **Principle of locality**
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
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.

  - How do caches take advantage of this?
Example: Locality?

```java
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **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**
Another Locality Example

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

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];

    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
Cost of Cache Misses

- 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”
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

- 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 hit times: 1 - 2 clock cycles for L1

- Miss Penalty
  - Additional time required because of a miss
  - Typically 50 - 200 cycles
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 for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ 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
Memory Hierarchies

■ Fundamental idea of a memory hierarchy:
  ▪ Each level k serves as a cache for the larger, slower, level k+1 below.

■ Why do memory hierarchies work?
  ▪ Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  ▪ Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

■ Big Idea: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.
An Example Memory Hierarchy

- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers
- Remote secondary storage (distributed file systems, web servers)
- Local secondary storage (local disks)
- Main memory (DRAM)
- On-chip L1 cache (SRAM)
- CPU registers

Smaller, faster, costlier per byte
Larger, slower, cheaper per byte

Caches - Memory Hierarchy
Intel Core i7 Cache Hierarchy

Processor package

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

Block size: 64 bytes for all caches.
Intel i7 Die

Memory Controller

Core

Queue

Shared L3 Cache
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
- Program optimizations that consider caches
Where should we put data in the cache?

Memory

Cache

- How can we compute this mapping?
Where should we put data in the cache?

Hmm.. The cache might get confused later! Why? And how do we solve that?
Use tags!

Cache Organization
What’s a cache block? (or *cache line*)

![Diagram showing cache block organization]

- **Byte Address**: The address of each byte in the cache block.
- **Block (line) number**: The block number to which each byte belongs.
- **Index**: The index used to map bytes to blocks.

Cache Organization
A puzzle.

- What can you infer from this:
  - Cache starts *empty*
  - Access (addr, hit/miss) stream
  - (10, miss), (11, hit), (12, miss)
Problems with direct mapped caches?

- What happens if a program uses addresses 2, 6, 2, 6, 2, ...?
**Associativity**

- What if we could store data in *any* place in the cache?
- But that might slow down caches... so we do something in between.

<table>
<thead>
<tr>
<th>Associativity</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
<td>1 set, 8 blocks</td>
</tr>
</tbody>
</table>

- **1-way**: 8 sets, 1 block each
- **2-way**: 4 sets, 2 blocks each
- **4-way**: 2 sets, 4 blocks each
- **8-way**: 1 set, 8 blocks

- **direct mapped**
- **fully associative**
But now how do I know where data goes?

Our example used a $2^2$-block cache with $2^1$ bytes per block. Where would 13 (1101) be stored?
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 011 0011.

```

<table>
<thead>
<tr>
<th>m-bit Address</th>
<th>(m-k-n) bits</th>
<th>k bits</th>
<th>n-bit Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- 1-way associativity
  - 8 sets, 1 block each
  - k = ?

- 2-way associativity
  - 4 sets, 2 blocks each
  - k = ?

- 4-way associativity
  - 2 sets, 4 blocks each
  - k = ?

Cache Organization
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...0110000 011 0011.
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, which one should we replace?
- Replace something, of course, but what?
  - Caches typically use something close to least-recently-used
Another puzzle.

- What can you infer from this:
  - Cache starts *empty*
  - Access (addr, hit/miss) stream
  - (10, miss); (12, miss); (10, miss)
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization (part 2)
- Program optimizations that consider caches
General Cache Organization (S, E, B)

- **S** = $2^s$ sets
- **E** = $2^e$ lines per set

**Cache size:**
$S \times E \times B$ data bytes

- **B** = $2^b$ bytes of data per cache line (the data block)

Valid bit

Set

Line
Cache Read

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

$E = 2^e$ lines per set

$S = 2^s$ sets

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

Address of byte in memory:
- $t$ bits: tag
- $s$ bits: set index
- $b$ bits: block offset

Data begins at this offset

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

S = 2^5 sets

Address of int:

find set
Example: Direct-Mapped Cache (E = 1)

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

Valid? + Match?: yes = hit

Address of int:

Block offset

Cache Organization
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:

| t bits | 0...01 | 100 |

Find set

Cache Organization
E-way Set-Associative Cache (Here: $E = 2$)

$E = 2$: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

$$\begin{array}{c}
\text{t bits} \\
\hline
0...01 \\
100
\end{array}$$

Compare both

valid? + match: yes = hit

Cache Organization
E-way Set-Associative Cache (Here: E = 2)

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

valid? + match: yes = hit

Address of short int:
\[ t \text{ bits} \quad 0\ldots01 \quad 100 \]

compare both

V   tag 0 1 2 3 4 5 6 7

V   tag 0 1 2 3 4 5 6 7

short int (2 Bytes) is here

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

Cache Organization
Types of Cache Misses

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

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block \( i \) must be placed in slot \( (i \mod \text{size}) \)), **direct-mapped**
    - if more than one, \( n \)-way **set-associative** (where \( n \) is a power of 2)
  - 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 (the **working set**) is larger than the cache (just won’t fit)
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What is the main problem with that?
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory

- **What to do on a write-hit?**
  - **Write-through** (write immediately to memory)
  - **Write-back** (defer write to memory until line is evicted)
    - Need a *dirty bit* to indicate if line is 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** (just write immediately to memory)

- **Typical caches:**
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Intel Core i7 Cache Hierarchy

Processor package

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

**Block size:** 64 bytes for all caches.
Memory and Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
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?**
  - Proper choice of algorithm
  - Loop transformations
Example: Matrix Multiplication

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 = 64 bytes = 8 doubles
- Cache size $C << n$ (much smaller than $n$)

First iteration:

- $n/8 + n = 9n/8$ misses (omitting matrix $c$)

Afterwards in cache: (schematic)
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C << n (much smaller than n)

Other iterations:
- Again:
  \[ \frac{n}{8} + n = \frac{9n}{8} \text{ misses} \]
  (omitting matrix c)

Total misses:
- \[ \frac{9n}{8} \cdot n^2 = \left(\frac{9}{8}\right) \cdot n^3 \]
Blocked Matrix Multiplication

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; i1++)
                        for (j1 = j; j1 < j+B; j1++)
                            for (k1 = k; k1 < k+B; k1++)
                                c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}

Block size B x B
Caches and Program Optimizations

\[ \text{c} = (\text{a} \ast \text{b}) \]
Cache Miss Analysis

Assume:

- Cache block = 64 bytes = 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
- $2n/B \times B^2/8 = nB/4$ (omitting matrix $c$)

Afterwards in cache (schematic)
Cache Miss Analysis

Assume:
- Cache block = 64 bytes = 8 doubles
- Cache size $C << n$ (much smaller than $n$)
- Three blocks fit into cache: $3B^2 < C$

Other (block) iterations:
- Same as first iteration
- $2n/B \times B^2/8 = nB/4$

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

■ No blocking: \((9/8) * n^3\)
■ Blocking: \(1/(4B) * n^3\)
■ If \(B = 8\) difference is \(4 * 8 * 9 / 8 = 36x\)
■ If \(B = 16\) difference is \(4 * 16 * 9 / 8 = 72x\)

■ Suggests 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
Cache-Friendly Code

- **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)
    - Focus on inner loop code
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

Caches and Program Optimizations