The Hardware/Software Interface
CSE351 Spring 2015
Lecture 16

Instructor:
Katelin Bailey

Teaching Assistants:
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
110000011111101000011111

Computer system:

OS:

Memory, data, & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Not to forget…

Why do caches work?

CPU

A little of super fast memory (cache$)

Managed entirely by hardware
No programmer control

cache

Lots of slower Mem
Back to the Core i7 to look at ways

Processor package

Core 0

Regs

L1 d-cache

L1 i-cache

L2 unified cache

L3 unified cache (shared by all cores)

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.
General Cache Mechanics

Cache

| 8 | 9 | 14 | 3 |

Memory

| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

Hit? Miss?

Smaller, faster, more expensive memory caches a subset of the blocks.

Data is copied in block-sized transfer units.

Larger, slower, cheaper memory viewed as partitioned into “blocks”
Where should we put data in the cache?

How can we compute this mapping?
- Address mod cache size
- Same as low-order $\log_2(\text{cache size})$ bits
Where should we put data in the cache?

Collision on green! Confusing!
Why does this happen? And what could we do to fix it?
Use tags to record which location is cached

tag = rest of address bits
What’s a cache block? (or cache line)

What’s the block/line size?

Typical block/line sizes: 32 bytes, 64 bytes
A puzzle.

- What can you infer from this:
  - Cache starts empty
  - Access (addr, hit/miss) stream:
  - (10, miss), (11, hit), (12, miss)

\[ \text{block size} \geq 2 \text{ bytes} \quad \text{block size} < 8 \text{ bytes} \]
Problems with direct mapped caches?

- **direct mapped:**
  - Each memory address can be mapped to exactly one index in the cache.

What happens if the program uses 2, 6, 2, 6?

- Conflict! Cache thrashing!
Associativity

- What if we could store data in any place in the cache?
What if we could store data in any place in the cache?
That might slow down caches (more complicated hardware), so we do something in between.
Each address maps to exactly one set.

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
Now how do I know where data goes?

m-bit Address

Tag

Index

(m-k-n) bits

k bits

n-bit Block Offset
What’s a cache block? (or \textit{cache line})

typical block/line sizes: 32 bytes, 64 bytes
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?

```
<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></td>
<td>Tag</td>
<td>Index</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>4-bit Address</th>
<th>? bits</th>
<th>? bits</th>
<th>?-bits Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Example placement in set-associative caches

- Where would data from address 0x1833 be placed?
  - Block size is 16 bytes.
- 0x1833 in binary is 00...01100000 011 0011.

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each
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-4) bits</th>
<th>k bits</th>
<th>4-bit Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tag</td>
<td>Index</td>
<td></td>
</tr>
</tbody>
</table>

- **k = 3**
  - 1-way associativity
  - 8 sets, 1 block each
  - Set 0

- **k = 2**
  - 2-way associativity
  - 4 sets, 2 blocks each
  - Set 0

- **k = 1**
  - 4-way associativity
  - 2 sets, 4 blocks each
  - Set 0

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?
Block replacement

- Replace something, of course, but what?

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each
Block replacement

- Replace something, of course, but what?
  - Obvious for direct-mapped caches, what about set-associative?
Block replacement

- Replace something, of course, but what?
  - Caches typically use something close to least recently used (LRU)
  - (hardware usually implements “not most recently used”)

```
1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each
```
Another puzzle.

• What can you infer from this:

• Cache starts empty
• Access (addr, hit/miss) stream

• (10, miss); (12, miss); (10, miss)

12 is not in the same block as 10
12’s block replaced 10’s block

direct-mapped cache
General Cache Organization (S, E, B)

- **E** = $2^e$ lines per set (we say “E-way”)
- **S** = 2^s sets
- **B** = $2^b$ bytes of data per cache line (the data block)

**cache size:**

$$S \times E \times B \text{ data bytes}$$
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
- s bits
- b bits

Tag set block

Index offset

data begins at this offset
Example: Direct-Mapped Cache (E = 1)

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

S = 2^s sets

Address of int:
- t bits: 0...01 100

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

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

Address of int:

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

valid? + match?: yes = hit

block offset
Example: Direct-Mapped Cache (E = 1)

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

Address of int:

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

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

Address of short int:

<table>
<thead>
<tr>
<th>t bits</th>
<th>0...10100</th>
</tr>
</thead>
</table>

Valid? + match: yes = hit

compare both

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

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

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

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

- **Conflict miss**
  - 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
    - **direct-mapped cache**s have more conflict misses than **n-way set-associative** (where n is a power of 2 and n $> 1$)

- **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’s the 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, all caches in between.
  - Write-back: defer write to memory until line is evicted (replaced)
    - 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 or reads 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
Write-back, write-allocate example

mov 0xFACE, T

Cache

Memory

0xBEEF

0xCAFE

0xBEEF

dirty bit
Write-back, write-allocate example

```
mov 0xFACE, T  mov 0xFEED, T  mov U, %rax
```

Cache

```
T  0xFEED
  1
```

dirty bit

Memory

```
T  0xCAFE
U  0xBEEF
```
Write-back, write-allocate example

mov 0xFACE, T  mov 0xFEED, T  mov U, %rax

Cache

<table>
<thead>
<tr>
<th>U</th>
<th>0xBEEF</th>
</tr>
</thead>
</table>

Memory

| T   | 0xFEED |
| U   | 0xBEEF |
Back to the Core i7 to look at ways

Processor package

Core 0
- Regs
  - L1 d-cache
  - L1 i-cache
- L2 unified cache
- L3 unified cache (shared by all cores)

Core 3
- Regs
  - L1 d-cache
  - L1 i-cache
- L2 unified cache
- L3 unified cache

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.

slower, but more likely to hit
Where else is caching used?
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 (for placement)
  - 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
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 can you achieve locality?
  • Proper choice of algorithm
  • Loop transformations
Example: Matrix Multiplication

```c
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];
}
```

$\begin{bmatrix}
    a & b \\
\end{bmatrix}
\begin{bmatrix}
    c \\
\end{bmatrix}
= 
\begin{bmatrix}
    (AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \\
\end{bmatrix}$

memory access pattern?
Assume:
• Matrix elements are doubles
• Cache block = 64 bytes = 8 doubles
• Cache size $C << n$ (much smaller than $n$, not left-shifted by $n$)

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:
    \[ n/8 + n = 9n/8 \text{ misses} \]
    (omitting matrix $c$)

• **Total misses:**
  - \[ 9n/8 \cdot n^2 = (9/8) \cdot n^3 \]
Blocked Matrix Multiplication

```c
double *c = 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

\[ c = (\text{double} \ *) \text{ calloc}(\text{sizeof}(\text{double}), \ n*\text{n}); \]

/* 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];
}
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)

B^2 elements per block, 8 per cache line

n/B blocks per row, n/B blocks per column

Afterwards in cache (schematic)
Cache Miss Analysis

• **Assume:**
  • Cache block = 64 bytes = 8 doubles
  • Cache size $C \ll n$ (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: \((9/8) \times n^3\)
- Blocking: \(1/(4B) \times n^3\)
- If B = 8, difference is \(4 \times 8 \times 9 / 8 = 36\times\)
- If B = 16, difference is \(4 \times 16 \times 9 / 8 = 72\times\)

  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
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.