Today

- Midterms back (at the end...)

- Homework 3 out later today

- How is Lab 3 going?
  - Due Wednesday

- Lecture and section *swapped* this Thursday/Friday.
- Just go to the usual rooms at the usual times.
  - Thursday in Loew 105: lecture
  - Friday in Johnson 022: section

- Back to $!
Not to forget...

Why do caches work?

managed entirely by hardware
no programmer control

Lots of slower Mem

A little of super fast memory (cache$)

CPU

cache
General Cache Mechanics

**Hit? Miss?**

**Cache**

| 8 | 9 | 14 | 3 |

Data is copied in block-sized transfer units

**Memory**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Larger, slower, cheaper memory viewed as partitioned into “blocks”

Smaller, faster, more expensive memory caches a subset of the 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.
Hmm.. The cache might get confused later!
Why? And how do we solve that?
Use tags to record which location is cached

Memory

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Cache

Index

00
01
10
11

Tag

Tag

Data

00
??
01
01

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

Byte Address  | Block (line) number
---|---
0 | 0
1 | 1
2 | 2
3 | 3
4 | 4
5 | 5
6 | 6
7 | 7
8 | 9
9 | 10
10 | 11
11 | 12
12 | 13
13 | 14
14 | 15

Index
0 1 2 3

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)

  block size $\geq 2$ bytes

  block size $< 8$ bytes
Problems with direct mapped caches?

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

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

- **conflict**
Associativity

- What if we could store data in *any* place in the cache?
Associativity

- 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.

<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>8 sets,</td>
<td>4 sets,</td>
<td>2 sets,</td>
<td>1 set,</td>
<td></td>
</tr>
<tr>
<td>1 block each</td>
<td>2 blocks each</td>
<td>4 blocks each</td>
<td>8 blocks</td>
<td></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*
Now how do I know where data goes?

m-bit Address

\[ \begin{array}{c}
\text{(m-k-n) bits} \\
\text{Tag} \\
\text{Index}
\end{array} \]

k bits

n-bit Block Offset
What’s a cache block? (or *cache line*)

![Diagram showing byte address, block (line) number, and index]

- **Byte Address**
- **Block (line) number**
- **Index**

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

**Block/line size = ?**

**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?
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.

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each

15
Example placement in set-associative caches

Where would data from address 0x1833 be placed?

- Block size is 16 bytes.

0x1833 in binary is 00...0110000011 0011.

16-byte block can be divided into

\[
\begin{array}{c|c|c}
(m-k-4) \text{ bits} & k \text{ bits} & 4\text{-bit Block} \\
\hline
\text{Tag} & \text{Index} & \text{Offset}
\end{array}
\]

\[k = 3\]

1-way associativity
8 sets, 1 block each

\[k = 2\]

2-way associativity
4 sets, 2 blocks each

\[k = 1\]

4-way associativity
2 sets, 4 blocks each
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?

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?

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?

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?
  - Caches typically use something close to least recently used (LRU)
  - (hardware usually implements “not most recently used”)

<table>
<thead>
<tr>
<th>Set</th>
<th>1-way associativity</th>
<th>2-way associativity</th>
<th>4-way associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>1</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>6</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>7</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
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:** $S \times E \times B$ data bytes
Cache Read

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

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

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

Address of byte in memory:
- \( t \) bits
- \( s \) bits
- \( b \) bits

- tag
- set index
- block offset

Valid bit

\[ B = 2^b \text{ bytes of data per cache line (the data block)} \]
Example: Direct-Mapped Cache (E = 1)

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

\[ S = 2^5 \text{ sets} \]

Address of int:
\[
\begin{array}{c}
\text{t bits} \\
0...01 \\
100
\end{array}
\]

find set

\[
\begin{array}{c|c}
\text{v} & \text{tag} \\
0 & 1234567 \\
1 & 234567 \\
2 & 01234567 \\
3 & 01234567 \\
4 & 01234567 \\
5 & 01234567 \\
6 & 01234567 \\
7 & 01234567
\end{array}
\]
Example: Direct-Mapped Cache (E = 1)

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

---

**Diagram:**
- **Valid?**
- **Match?: yes = hit**
- **Address of int:**
  - **t bits:** 0...1 100
  - **block offset**

---

**Table:**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
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
Assume sum, i, j in registers
Address of an aligned element of a: aa...ayyyxxxx000
Assume: cold (empty) cache
3 bits for set, 5 bits for offset

\[ \text{aa...ayyy yxx xx000} \]

0,0: aa...a000 000 00000

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
</tr>
<tr>
<td>0,4</td>
<td>0,5</td>
<td>0,6</td>
<td>0,7</td>
</tr>
<tr>
<td>0,8</td>
<td>0,9</td>
<td>0,a</td>
<td>0,b</td>
</tr>
<tr>
<td>0,c</td>
<td>0,d</td>
<td>0,e</td>
<td>0,f</td>
</tr>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
</tr>
<tr>
<td>1,4</td>
<td>1,5</td>
<td>1,6</td>
<td>1,7</td>
</tr>
<tr>
<td>1,8</td>
<td>1,9</td>
<td>1,a</td>
<td>1,b</td>
</tr>
<tr>
<td>1,c</td>
<td>1,d</td>
<td>1,e</td>
<td>1,f</td>
</tr>
</tbody>
</table>

32 B = 4 doubles
4 misses per row of array
4*16 = 64 misses
every access a miss
16*16 = 256 misses

Example (for E = 1)

```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;
}
```
Example (for \( E = 1 \))

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;
    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

In this example, cache blocks are 16 bytes; 8 sets in cache

- How many block offset bits?
- How many set index bits?

Address bits: ttt....t sss bbbb

- \( B = 16 = 2^b \): \( b=4 \) offset bits
- \( S = 8 = 2^s \): \( s=3 \) index bits

0: 000....0 000 0000
128: 000....1 000 0000
160: 000....1 010 0000

if \( x \) and \( y \) have aligned starting addresses,
e.g., \&\( x[0] = 0 \), \&\( y[0] = 128 \)

if \( x \) and \( y \) have unaligned starting addresses,
e.g., \&\( x[0] = 0 \), \&\( y[0] = 160 \)
E-way Set-Associative Cache (Here: \( E = 2 \))

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

Address of short int:

\[
\begin{array}{c|c|c}
\text{t bits} & 0\ldots01 & 100 \\
\end{array}
\]
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:

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

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

- **valid?** + **match: yes = hit**
- **compare both**

Address of short int:

- t bits
- 0...01
- 100

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), ...

Example (for E = 2)

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0;
    int i;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```

If \(x\) and \(y\) have aligned starting addresses, e.g. \&x[0] = 0, \&y[0] = 128, can still fit both because two lines in each set
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* caches 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 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, 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

why?
Write-back, write-allocate example

mov 0xFACE, T

Cache

U 0xBEEF 0

dirty bit

Memory

T
0xCAFE

U
0xBEEF
Write-back, write-allocate example

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

Cache

Memory

T

0xCAFE

U

0xBEEF

T

0xFEED

dirty bit
Write-back, write-allocate example

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

Cache

| U | 0xBEEF | 0 |

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 (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.

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];
}
```

\[
(AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj}
\]

memory access pattern?
Cache Miss Analysis

- **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:**
  - $\frac{n}{8} + n = \frac{9n}{8}$ misses (omitting matrix $c$)

- Afterwards **in cache:**
  - (schematic)

Spatial locality: chunks of 8 items in a row in same cache line

Each item in column in different cache line

First iteration:
- $\frac{n}{8} + n = \frac{9n}{8}$ misses (omitting matrix $c$)
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:**
  - $$\frac{9n}{8} \times n^2 = \left(\frac{9}{8}\right) \times n^3$$

  once per element
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
Cache Miss Analysis

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

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: $(9/8) \times n^3$
- Blocking: $1/(4B) \times n^3$
- If $B = 8$ difference is $4 \times 8 \times 9 / 8 = 36x$
- If $B = 16$ difference is $4 \times 16 \times 9 / 8 = 72x$

- 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
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

... (shared by all cores)

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.
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