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 $!

General Cache Mechanics

Hit? Miss?

Data is copied in block-sized transfer units

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

Cache

Memory

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

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

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

Where should we put data in the cache?

Memory

Cache

00 01 10 11

Index

Data

address mod cache size

same as

low-order log₂(cache size) bits

How can we compute this mapping?
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

tag = rest of address bits

A puzzle.

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

  block size >= 2 bytes  block size < 8 bytes

What’s a cache block? (or cache line)

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

```
Memory Address
0000
0001
0010
0100
0101
0110
1000
1001
1010
1011
1100
1101
1110
1111
```

Index

```
00
01
10
11
```

Associativity

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

Now how do I know where data goes?

- 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

```
Set 0
- Set 0
  - Set 0
    - Set 0
      - Set 0
        - Set 0
          - Set 0

Set 1
- Set 1
  - Set 1
    - Set 1
      - Set 1
        - Set 1

Set 2
- Set 2
  - Set 2
    - Set 2
      - Set 2

Set 3
- Set 3
  - Set 3

Set 4
- Set 4

Set 5
- Set 5

Set 6
- Set 6

Set 7
- Set 7
```

- **direct mapped**

- **fully associative**
What's a cache block? (or cache line)

block/line size = ?

typical block/line sizes: 32 bytes, 64 bytes

Now how do I know where data goes?

m-bit Address

Tag  Index  n-bit Block Offset

Our example used a 2^2-block cache with 2^1 bytes per block. Where would 13 (1101) be stored?

4-bit Address

? bits  ? bits  f-bits Block Offset

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.

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?

1-way associativity
8 sets, 1 block each

2-way associativity
4 sets, 2 blocks each

4-way associativity
2 sets, 4 blocks each

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

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

Replace something, of course, but what?
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 line (the data block)

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

Address of int:

0 1 2 3 4 5 6 7

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

<table>
<thead>
<tr>
<th>Valid?</th>
<th>Match?: y e s = h i t</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

Address of int: t bits 0..01 100

Block offset

No match: old line is evicted and replaced

Example (for E = 1)

Assume sum, i, j in registers
Address of an aligned element of a: aa...ayy yyy xx000

```
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
3 bits for set, 5 bits for offset
aa...ayy yyy xx000 0,0: aa...a000 000 00000

```
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: tt...t s s s b b b b
B = 16 = 2^4: b=4 offset bits
S = 8 = 2^3: s=3 index bits
0: 0000...0 000000
128: 0000...1 000000
160: 0000...1 010000

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

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

---

**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

---

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** 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 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
  - Why?
  - Write-through + No-write-allocate, occasionally

Write-back, write-allocate example

```
mov 0xFACE, T
```

<table>
<thead>
<tr>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x0BEEF</td>
<td>0x0BEEF</td>
</tr>
<tr>
<td>dirty bit</td>
<td>0xCAFE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0x0BEEF</td>
</tr>
<tr>
<td>0x0CAFE</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>T</td>
</tr>
</tbody>
</table>
Write-back, write-allocate example

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

Memory

```
T | 0xFACE
U | 0xBEFF
```

Cache

```
T | 0xFEEF 1
```
dirty bit

Write-back, write-allocate example

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

Memory

```
T | 0xFEFF
U | 0xBEFF
```

Cache

```
U | 0xBEFF 0
```
dirty bit

Back to the Core i7 to look at ways

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: 3 cycles

L2 unified cache:
- 256 KB, 8-way,
  Access: 4 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

Example: Matrix Multiplication

```c
#include <stdio.h>

int main()
{
    int i, j, k;
    double a[100][100];
    double b[100][100];
    double c[100][100];

    for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
            for (k = 0; k < 100; k++)
                c[i][j] += a[i][k] * b[k][j];

    return 0;
}
```

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

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:**
  - \(n/8 + n = 9n/8\) 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

\(n/8\) misses

n misses

8 wide
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:
  - \( \frac{n}{8} + n = 9n/8 \) misses
  - (omitting matrix C)

Total misses:
- \( 9n/8 \cdot n^2 = (9/8) \cdot n^3 \) once per element

Block size B x B

Block size B x 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
- \( 2n/B \cdot B^2/8 = nB/4 \)
  - (omitting matrix C)
  - \( n/B \) blocks per row, \( n/B \) blocks per column
- Afterwards in cache (schematic)

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

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

- No blocking: \( \frac{9}{8} \times n^3 \)
- Blocking: \( \frac{1}{4B} \times n^3 \)
- If \( B = 8 \) difference is \( 4 \times 8 = 32x \)
- If \( B = 16 \) difference is \( 4 \times 16 = 72x \)

- 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
  - 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.

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